Work > Book 1

Introductory Activities for the Ab Initio Developer

May 2011 > Part No. AB1689

Ab Initio Software LLC
201 Spring Street > Lexington MA 02421
Voice +1 781.301.2000 > Fax +1 781.301.2001 > support@abinitio.com

INTELLECTUAL PROPERTY RIGHTS & WARRANTY DISCLAIMER

CONFIDENTIAL & PROPRIETARY

This document is confidential and a trade secret of Ab Initio. This document is furnished under a license and may be used only in accordance with the terms of that license and with the inclusion of the copyright notice set forth below.

COPYRIGHTS

TRADEMARKS

The following are worldwide trademarks or service marks of or licensed to Ab Initio (those marked ® are registered in the U.S. Trademark Office, and may be registered in other countries):

>®	Conduct>It®	EME Portal®	Meta>Operating System®
Ab Initio®	Continuous Flows®	Engine by Ab Initio®	Meta OS®
Ab Initio I>O®	Continuous>Flows®	Enterprise Meta>Environment®	Meta>OS®
Abinitio.com®	Cooperating Enterprise®	Enterprise Metadata Environment®	Plan>It®
BRE®	Cooperating System®	Enterprise MetaEnvironment®	Re>Posit®
Co>Operating Enterprise®	Cooperating®	GDE®	Re>Source®
Co>Operating System®	Data>Profiler®	Graphical Development Environment	Server++®
Co>Operating®	Director®	Graph It®	Server+Server®
Co>Operation®	Dynamic Data Mart®	Graph>It®	Shop for Data®
Co>Operative®	E2E®	I>O®	The Company Operating System®
Co>OpSys®	EME®	Init.com	
Co>Ordinate®	EME Desktop Portal®	INIT®	
Co>Ordinator®	EME Management Console®	Meta Operating System®	

Certain product, service, or company designations for companies other than Ab Initio are mentioned in this document for identification purposes only. Such designations are often claimed as trademarks or service marks. In instances where Ab Initio is aware of a claim, the designation appears in initial capital or all capital letters. However, readers should contact the appropriate companies for more complete information regarding such designations and their registration status.

RESTRICTED RIGHTS LEGEND

If any Ab Initio software or documentation is acquired by or on behalf of the United States of America, its agencies and/or instrumentalities (the "Government"), the Government agrees that such software or documentation is provided with Restricted Rights, and is "commercial computer software" or "commercial computer software documentation." Use, duplication, or disclosure by the Government is subject to restrictions as set forth in the Rights in Technical Data and Computer Software provisions at DFARS 252.227-7013(c)(1)(ii) or the Commercial Computer Software – Restricted Rights provisions at 48 CFR 52.227-19, as applicable. Manufacturer is Ab Initio Software LLC, 201 Spring Street, Lexington, MA 02421.

WARRANTY DISCLAIMER

THIS DOCUMENT IS SUBJECT TO CHANGE WITHOUT NOTICE. AB INITIO MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. AB INITIO SHALL NOT BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR INCIDENTAL OR CONSEQUENTIAL DAMAGE IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL.

Contents

1

Using *Work>Book 1*

The goal of *Work>Book 1* is to help you attain a basic working knowledge of the scope and abilities of Ab Initio software. After completing the *Intro Course*, you should use *Work>Book 1* to review what you have learned, reinforce your skills, and deepen your understanding. The book is structured as follows:

- **Chapter 1** provides an overview of development practices you should follow when using Ab Initio software in general and *Work>Book 1* in particular. It also summarizes the different types of Ab Initio components and describes the imaginary business setting and data model used for the activities and business problems presented later in the book.
- **Chapters 2 through 4** describe how to set up the sandbox environment for the activities, add the needed datasets, and view data.

- **Chapters 5 through 19** each focus on one frequently used Ab Initio component or feature, and present activities for learning how to use it to solve common business problems. Each chapter begins with general information and helpful insights about using the component or feature. Following these are guidelines for solving the problems posed in the activities, and at the end of the activity descriptions is a pointer to the solutions.
- **Chapter 20** poses a set of business problems that might occur in the business setting used for the activities in earlier chapters, and offers hints on how to solve them.

This first chapter describes how to use *Work>Book 1* effectively. It covers the following topics:

Good development practices

This section describes development practices you should follow when using Ab Initio software in general and *Work>Book 1* in particular.

SOLVING PROBLEMS

A core principle underlying Ab Initio software is that hard problems become easier when they are broken down into simple steps. No matter how complex the business process they address, all Ab Initio applications are built from a handful of simple building blocks. You should approach each

problem by breaking it down into a sequence of simpler steps. Each of these steps might need to be broken down further before you reach the level of individual graph components. *Work>Book 1* introduces many simple graphs that will become pieces of larger graphs.

KNOWING YOUR DATA

Knowing your data is a cornerstone of good development practice. You should continually strive to learn and understand both the overall intended characteristics and the unintended peculiarities of the data you will process. What fields are in which datasets? Which are keys? What values can they have? At what level of aggregation do they occur? What kinds of data quality issues (bad or missing values, duplicate records, referential integrity problems, and so on) are present? Armed with the answers to these questions, you will be able to use the data to solve complex business problems quickly and effectively. If you know your data, you will encounter many fewer surprises while developing, and the applications you build will be far more robust.

USING SAMPLE TEST DATA

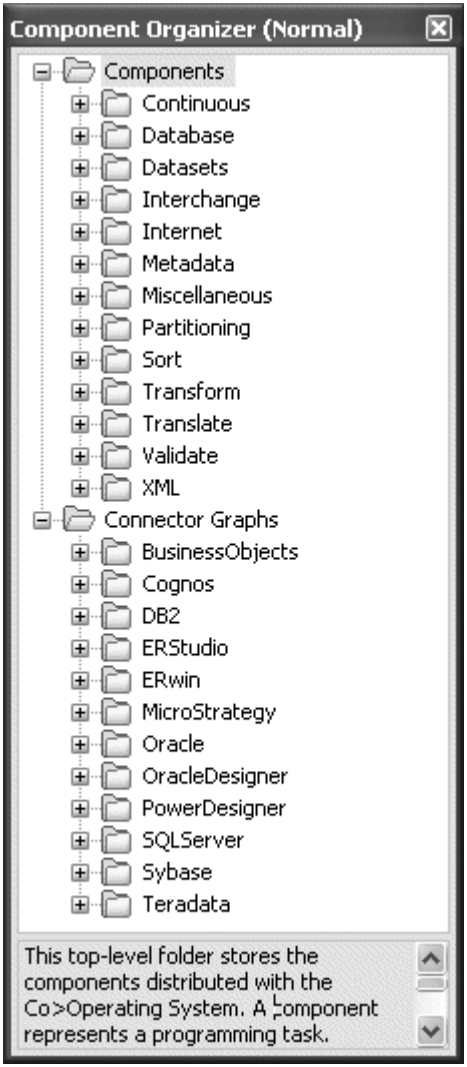
Ab Initio software lets you maintain a tight connection between development and testing, which in turn enables rapid development. Although Ab Initio software makes it feasible to process vast amounts of data in a comparatively short time, it is rarely necessary to continually test against large amounts of data during development, and doing so might be an inefficient use of your time. Instead, you should develop and test against small but representative samples of data until your graph is complete and produces the expected results from the test data. This test data should, to the greatest extent possible, capture the interesting cases that are likely to arise in real data — including data quality issues — with a minimum of redundancy. You might find it helpful or even necessary to create some test data by hand. Knowing your data, of course, is a requirement to producing test data.

KEEPING A RECORD OF YOUR WORK

New users of the Ab Initio Graphical Development Environment™ (GDE®) are often tempted to do all their work in a single graph. As you progress through *Work>Book 1*, you will frequently find that a previous graph can be modified to address the current question. We recommend that instead of simply modifying an existing graph, you save it with a new name for each activity. By following this practice, you will have a complete record of your work and a richer set of examples to refer to when you are doing the later activities.

Components

The **Component Organizer** (shown next) contains all the parts used to assemble Ab Initio graphs. A small group of these parts forms a core subset of frequently used components that are the subject of *Work>Book 1*.



The core subset covered in *Work>Book 1* can be divided into smaller groups as follows:

- **Dataset components** — Provide read and write access to persistent data. They include INPUT FILE, OUTPUT FILE, INTERMEDIATE FILE, and LOOKUP FILE, and are located in the **Datasets** folder of the **Component Organizer**.
- **Filtering and sorting components** — Change the order of records in a flow or determine which flow a given record traverses, but do not alter the contents of individual records. The FILTER BY EXPRESSION and DEDUP SORTED components are in the **Transform** folder, and the SORT and SORT WITHIN GROUPS components are in the **Sort** folder.
- **Transforming components** — Affect the contents of records. For example, they allow you to manipulate the fields in a record (REFORMAT), aggregate values across records in a group (ROLLUP), and combine the contents of records from multiple flows (JOIN). These components are in the **Transform** folder.
- **Partitioning and departitioning components** — Affect the parallelism of data, sending records to different partitions (PARTITION BY ROUND-ROBIN and PARTITION BY KEY) or receiving data from different partitions (GATHER and MERGE). The PARTITION BY ROUND-ROBIN and PARTITION BY KEY components are in the **Partition** folder; the GATHER and MERGE components are in the **Departition** folder.
- **Miscellaneous components** — Provide basic "plumbing" parts for graphs. The TRASH component simply discards its input data. The REPLICATE component lets you share the results of a component with more than one downstream component. Both TRASH and REPLICATE are in the **Miscellaneous** folder.

Activities

In this book, an *activity* is a hands-on exercise that you do with the Ab Initio GDE. Each activity is described in a numbered blue paragraph, and the numbers map to the names of the solution graphs. See "Checking your graphs and output data against the solutions" on page 19.

Business setting

All the activities and business problems in *Work>Book 1* are based on a single retail business scenario. As you work through the activities, you will become familiar with the data model, and increasingly with the data itself. Coming to know your data will benefit you in several ways:

- It reduces the "learning curve" aspect of problem solving, as you will not have to learn and understand a new business situation to tackle each new problem. This simulates the real-life experience of most Ab Initio developers, who work for an extended period on a project in a business that they know or come to know well.
- You will become adept at addressing data quality issues. All realistic data has such issues, and the data in *Work>Book 1* is no exception. Data quality issues can arise in every stage of a project and must be addressed.
- You will develop a feel for how to work with the data, and you will know which fields you are interested in and how to get to them. This in turn will make it easier to solve increasingly complex business problems. You will spend less energy on details, such as finding the location of a specific piece of information, and have more energy to focus on solving real problems.

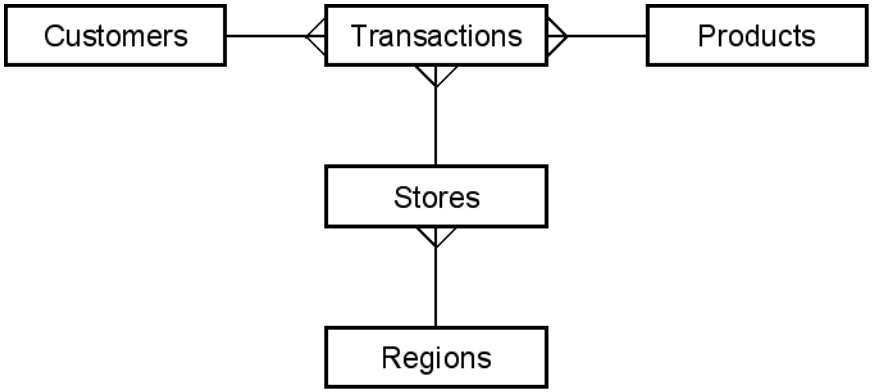
Work>Book 1 is based on the following imaginary business situation: a company named DIY ("Do It Yourself!") has a chain of approximately 200 stores across the United States. The stores are in 50 cities throughout 10 states, which are parceled into five geographic regions. Your job is to help DIY answer questions about the home improvement department in its stores. At your disposal are several datasets that form part of an enterprise-wide data warehouse.

The *Work>Book 1* data model

The *Work>Book 1* activities are based on a data model involving the following five datasets:

- **Transactions** — Contains records of purchases in the home improvement department in the years 1999 and 2000. Original store invoices were previously normalized into individual line items to produce the **Transactions** dataset, so there is a separate transaction record for each product purchased. Details of the fields in the **Transactions** dataset will be examined more closely later; for now, it is enough to note that a transaction record includes a product code, the quantity purchased, and the transaction amount, in addition to a customer identifier and a store number. The dataset has one key field, **transaction_id**.
- **Customers** — Contains the name and address of store customers. This dataset has one key field, **customer_id**.
- **Products** — Is an inventory file of products in the home improvement department and has one key field, **product_cd** (product code).
- **Stores** — Contains the address information for each store and has one key field, **store_no**.
- **Regions** — Is keyed by **store_no** and provides the **region_code** for each store.

The following figure shows the entity-relationship diagram for the *Work>Book 1* data model:



2

Setting up the environment

This chapter contains the following sections about setting up the *Work>Book 1* environment:

Installing and opening the *Work>Book 1* sandbox

After you have your settings configured so that the GDE is connected to the Ab Initio Co>Operating System®, you will need a place to hold the graphs and datasets you will produce. This workspace is called a *sandbox*, and this section describes how to install and open your *Work>Book 1* sandbox.

NOTE: The *Work>Book 1* sandbox will be installed in the default directory of the current host connection.

▶ **To install and open your *Work>Book 1* sandbox:**

1. From the GDE main menu, choose **Help > Examples**.
2. In the **Install Examples** dialog, select **Work>Book 1**.
3. Verify that the displayed host connection and installation path to the default directory are correct.

   If necessary, choose **Settings > Manage Connections** and click **Edit** to open the **Host Connection Settings** dialog and make changes.
4. Click **Install**.
5. On the **Examples** tab of the **Application Output** window, monitor the progress of the installation.

NOTE: You can click **Cancel** at any time to cancel the installation.

6. When the installation is complete, click **Open Sandbox**.

Parameters

A *parameter* is a named placeholder for a value to be supplied later. Parameters let you use logical names instead of actual values, thereby greatly enhancing the flexibility and portability of the graphs you develop. Parameters can be used in components, graphs, subgraphs, and sandboxes.

You usually access the value of a parameter by prefixing its name with a dollar sign (**$**). For example, **PROJECT_DIR** is the name of the parameter that gives the location of the current sandbox, and you write **$PROJECT_DIR** to get the actual location. You can define parameter values in terms of other parameters. Directories in a sandbox, such as the **dml** subdirectory, are usually referred to via parameters with names like **AI_DML**, which in turn have definitions like **$PROJECT_DIR/dml**.

You can view a sandbox's parameters by choosing **Project > Edit Sandbox > Parameters**. Different interpretations cause the parameter definition to be processed differently. For details on interpretation types, search for **"parameter interpretation"** in Ab Initio Help.

At the graph and sandbox level, using parameters facilitates portability. Graphs developed in one environment on one computer should run in another environment on the same computer or a different computer without direct modification of the graph. Using sandbox parameters achieves this by isolating the graph from environment-specific details such as where directories are located or what filenames to use.

Parameters make development easier and improve communication among developers. Parameter names for directories are easier to remember, shorter to type, and less awkward to communicate than complicated filesystem paths. They also promote uniform standards across projects and development efforts.

Parameters also strengthen the flexibility and adaptability of graphs. Changing the value of a parameter is easy and occurs in one place. In contrast, finding and changing all instances of a hardcoded value is awkward and error prone. For example, if the number of regions in your organization changed from five to seven, you would find it much easier to change the value of a **NUM_REGIONS** parameter from **5** to **7** than to identify only the occurrences of the constant **5** that refer to the number of regions and change them to **7**.

BEST PRACTICE

Use sandbox parameters.

Use sandbox parameters in all filesystem paths. Hardwired, absolute pathnames invariably hinder graph portability and reuse. By using sandbox parameters, you can ensure that graphs will be easy to migrate to new environments or adapt to new uses.

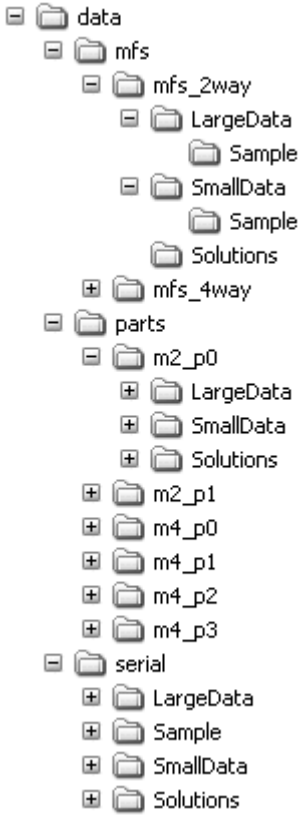
Sandbox parameters

Your **workbook1** sandbox defines several parameters for referring to different filesystem directories, including the following:

PARAMETER	VALUE	EXPLANATION
PROJECT_DIR	*sandbox-installation-directory*	Root directory of the sandbox; this is set when the sandbox is created.
TEST_FLAG	**SmallData** or **LargeData**	Version of the data to use; see page 17.
AI_DML	$PROJECT_DIR/dml	Directory for files containing record format descriptions (**.dml** files).
AI_XFR	$PROJECT_DIR/xfr	Directory for transform code (**.xfr** files).
DATA_MOUNT	$PROJECT_DIR/data	Central location for all data directories.
AI_SERIAL	$DATA_MOUNT/serial/$TEST_FLAG	Serial data area (depends on **$TEST_FLAG**).
AI_MFS_CONTROL	$DATA_MOUNT/mfs	Multifile directories. In the sandbox environment for *Work>Book 1*, this directory contains two subdirectories: **mfs_2way** (a two-way multifile system) and **mfs_4way** (a four-way multifile system).

PARAMETER	VALUE	EXPLANATION
AI_MFS	$AI_MFS_CONTROL/mfs_2way/$TEST_FLAG	Two-way multifile system (always referred to as **$AI_MFS** in graphs; depends on **$TEST_FLAG**).
SOL_SERIAL	$DATA_MOUNT/serial/Solutions	Serial data area for solution graph output.
SOL_MFS	$AI_MFS_CONTROL/mfs_2way/Solutions	Two-way multifile system for solution graph output.
SOL_WIDE_MFS	$AI_MFS_CONTROL/mfs_4way/Solutions	Four-way multifile system for solution graph output.

The following figure shows the structure of the data directory referred to by the **DATA_MOUNT** parameter:



▶ **To view the resolved values of sandbox parameters:**

1. If necessary, open a graph in the sandbox so that the **Edit** menu is available.
2. Choose **Edit > Parameters**.

3. At the top of the left pane of the **Parameters Editor**, select the sandbox.
4. In the top right pane of the **Parameters Editor**, do one of the following:
   - Let the mouse hover over the parameter of interest and view its resolved value in the pop-up window.
   - Select the parameter of interest and view its resolved value in the **Resolved value** pane at the bottom right of the **Parameters Editor**.

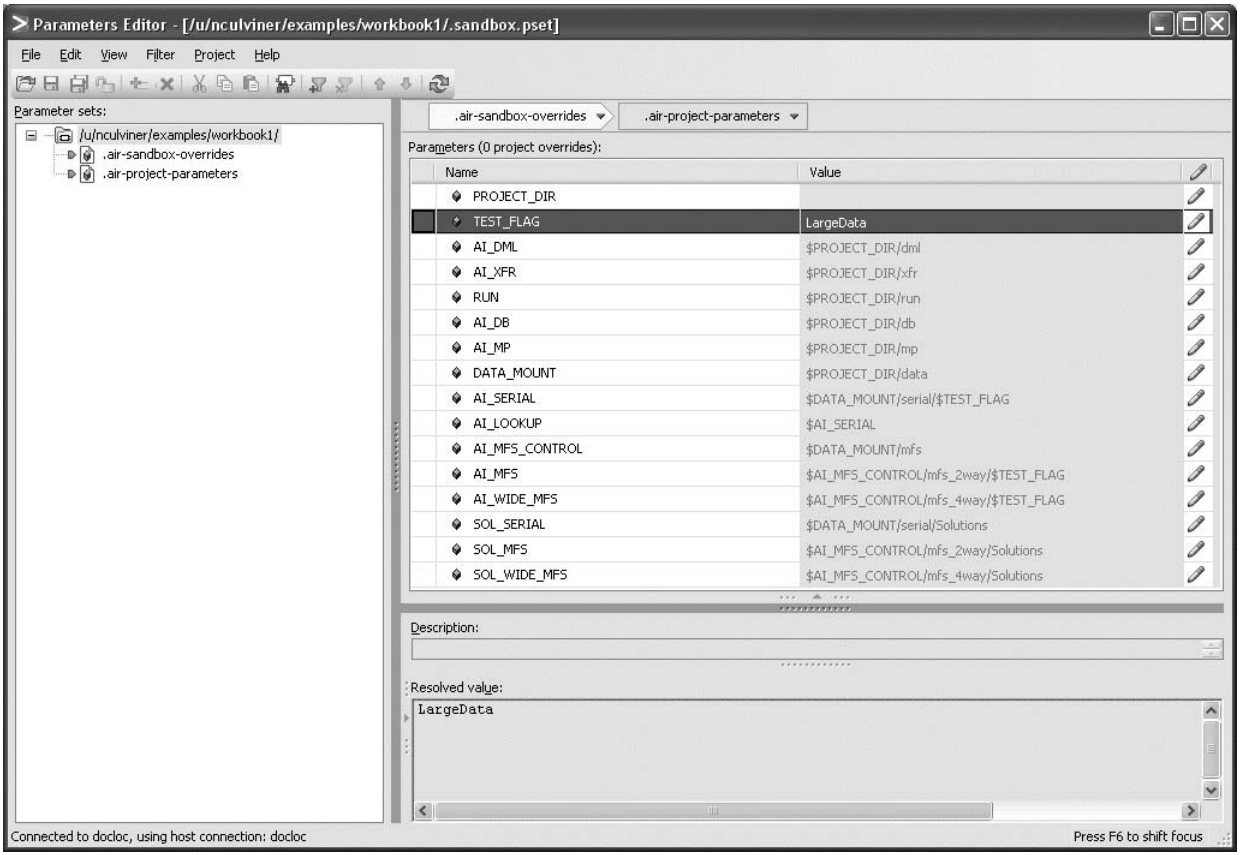
Small and large datasets

The sandbox is configured with two versions of data: a small version with which you will be doing most of your development and testing, and a large version that you will run against after your graphs work well with the small data. The parameter **TEST_FLAG** lets you switch easily between the two, so that you can run a graph against the small version or the large version without changing the graph itself. The parameter's value (**SmallData** or **LargeData**) is used as part of the pathname of the directories containing the input and output files for your graphs. The initial value is **SmallData**. (The third possible value, **Repository**, is reserved for the Ab Initio Enterprise Meta>Environment® source control system, and you should not use it when running graphs.)

PRACTICAL NOTE

Using the small and large datasets

The small dataset (**TEST_FLAG** = **SmallData**) is helpful for many of the activities, but to get meaningful output for others, you may have to use the large dataset (**TEST_FLAG** = **LargeData**) or create a different subset of it. To do this, you can use a LEADING RECORDS or FILTER BY EXPRESSION component.



Using sandbox parameters in the shell

You can easily make sandbox parameters available for use in the shell; simply execute the following command in the sandbox directory:

```
. ./ab_project_setup.ksh $(pwd)
```

After doing this, you can use shell commands that refer to the sandbox parameters. For example, you can use **cd $AI_DML** to change your current directory to the DML directory.

Checking your graphs and output data against the solutions

After creating and running any graph in the "Activities" section of each chapter, you can check your graph against the corresponding solution graph, and your output data against the solution graph's output data.

- The solution graphs are in **$PROJECT_DIR/solutions**.
- The solution output data is in the directories specified by the parameters **SOL_SERIAL**, **SOL_MFS**, and **SOL_WIDE_MFS** (see "Sandbox parameters" on page 14). Each filename ends with **_<#>.mp**, where **<#>** is the activity number.

3

Configuring datasets

Working with datasets should be almost second nature to you after completing *the Intro Course*. This chapter provides a quick review of the different kinds of datasets and introduces you to the data you will be using in *Work>Book 1*.

Data is at the heart of any business. For the purposes of *Work>Book 1*, your business is the imaginary retail company DIY, described in Chapter 1. You will be dealing with data such as transaction records, customer lists, and product information, and you will get valuable information about your business by processing it. First, though, you must be able to access your data. Dataset components provide that access. There are several kinds of dataset components, including:

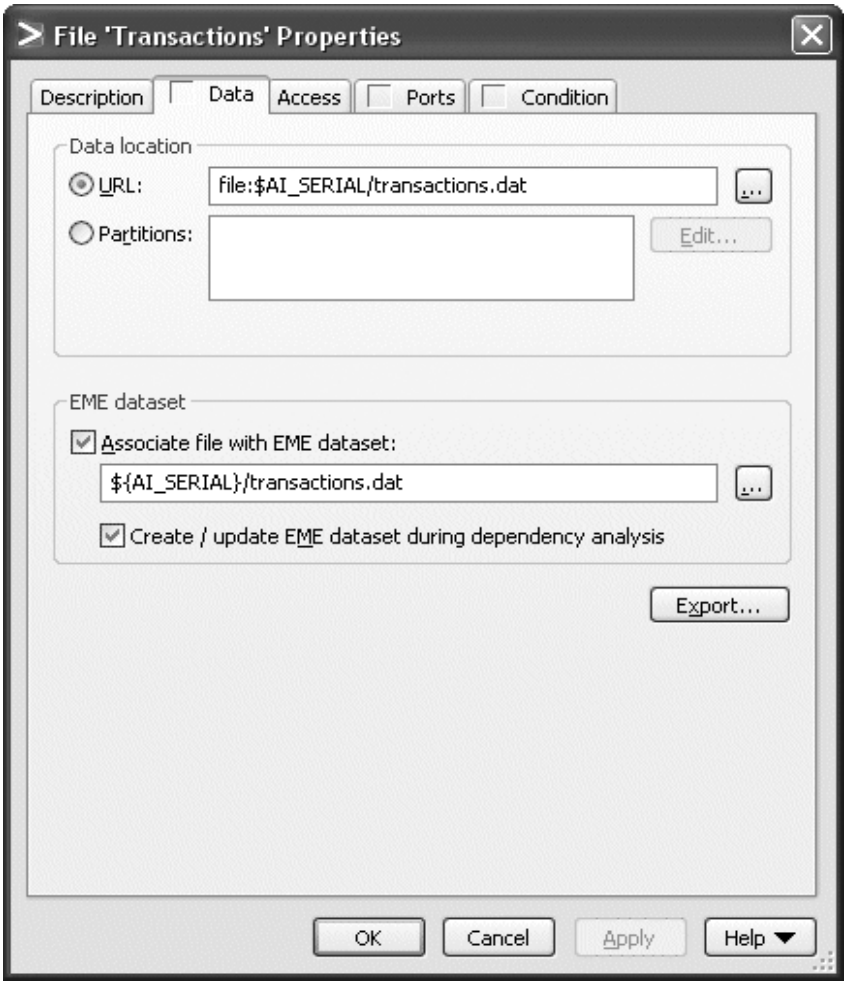
- The INPUT FILE component, which reads data records from a serial file or a multifile in the filesystem.

- The OUTPUT FILE component, which writes data records to a serial file or a multifile in the filesystem.
- The INTERMEDIATE FILE component, which can be used to write data records to a file in the middle of a graph. Writing data to disk in this way makes the records on that flow available for viewing after execution. This is useful for debugging and for future processing of intermediate data, either as an INPUT FILE in another graph or as a LOOKUP FILE in the current graph or another graph. (For information on lookup files, see Chapter 13.) For debugging, an alternative to an INTERMEDIATE FILE is a watcher on a flow.

The INPUT FILE, OUTPUT FILE, and INTERMEDIATE FILE components are in the **Datasets** folder of the **Component Organizer**.

Configuring an INPUT FILE component

To configure an INPUT FILE component, you must supply both a URL (file path) on the **Data** tab and a record format (described in detail in Chapter 10) on the **Ports** tab.



Activities

▶ **To begin these activities:**

1. Start the GDE.
2. Open a new graph.
3. Save the graph as **Datasets.mp** in the **mp** directory of your **workbook1** sandbox as follows:
   a. Choose **File > Save**.
   b. In the **Save As [Host]** dialog, select (if necessary) the **Host** radio button.
   c. In the **Host** drop-down list, select the host connection you used to install the **workbook1** sandbox (see page 12).
   d. Navigate to the **mp** directory of the **workbook1** sandbox and click **Save**.

   The lower-right corner of the status bar in the GDE window displays the name of the current sandbox. If the current graph is not associated with a sandbox, the status bar displays **No Sandbox**.

   This graph will contain the datasets needed for most of the activities in *Work>Book 1*.

BEST PRACTICE
Use descriptive labels.

For every component you add to a graph, set the label to something that describes its function, meaning, or purpose. By default, the label for any component conveys only the type of that component. For dataset components, choose a label that indicates what the data is or what it will be used for.

Adding the *Work>Book 1* datasets to the **Datasets.mp** graph

▶ **To insert an INPUT FILE component:**

1. Choose **Insert > Input File** and double-click the inserted component to open the **Properties** dialog.
2. On the **Description** tab of the **Properties** dialog, set the label for the dataset.
3. On the **Data** tab, specify the location of the data:

- If your data is in a multifile, enter **mfile:$AI_MFS** in the **URL** field and click ... to browse for the appropriate file. (Multifiles are parallel datasets and will be described in more detail in Chapter 16.)
- If your data is in a serial file, enter **file:$AI_SERIAL** in the **URL** field and click ... to browse for the appropriate file.

4. On the **Ports** tab, set the record format to be associated with the port of the dataset:
   a. Select **Use File**.
   b. In the **File** text box, enter **$AI_DML** and click ... to browse the DML directory for the appropriate file.

1. Insert an INPUT FILE component. Open the component properties, change the label to **Customers**, and specify the URL **mfile:$AI_MFS/customers.dat**. Click the **Ports** tab, click **Use file**, enter **$AI_DML** in the **File** text box, click **...**, and select **customers.dml**.
2. Insert an INPUT FILE component called **Transactions**, with URL **mfile:$AI_MFS/transactions.dat** and record format **$AI_DML/transactions.dml**.
3. Insert an INPUT FILE component called **Products**, with URL **file:$AI_SERIAL/products.dat** and record format **$AI_DML/products.dml**.
4. Insert an INPUT FILE component called **Stores**, with URL **file:$AI_SERIAL/stores.dat** and record format **$AI_DML/stores.dml**.

Renaming a copied component

If you paste a component into a graph containing another component with the same label, the GDE appends a number to the label of the pasted component to make the name unique.

5. Copy and paste the **Stores** dataset and modify its properties to rename it **Regions**, using the new URL **file:$AI_SERIAL/regions.dat** and record format **$AI_DML/regions.dml**.

6. Choose **File > Save** to save the changes you have made to the graph.

SOLUTION

The solution graph is **solutions/Datasets.mp**.

4

Viewing data

View Data is a powerful tool for inspecting records in a file. With it, you can quickly and easily view any range of records, in any or all partitions, in the format you want.

View Data options

To inspect the records in a file, right-click the component and choose **View Data** from the pop-up menu. The **View Data Options** dialog appears:

PRACTICAL NOTE

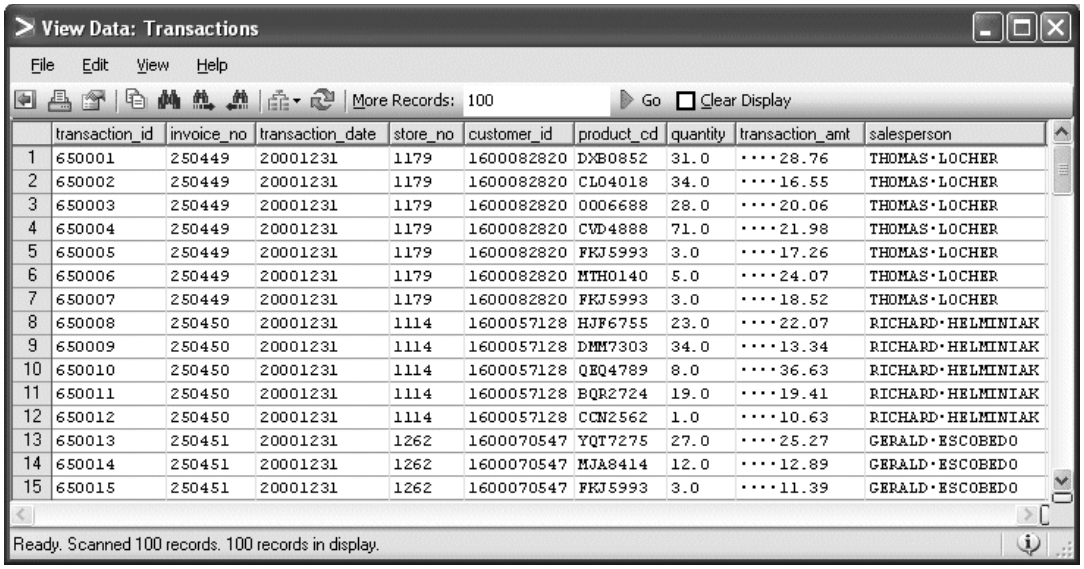
Using View Data to check an input file format

You can use **View Data** after creating an INPUT FILE component to quickly verify that both the URL to the data and its associated record format are correct.

Use the dialog's tabs to specify which records you want to display and the format to use (by default, a subset of the data will be displayed):

- On the **Records** tab, specify the start record, the number of records you want to display, and the partition (if the dataset refers to a multifile).
- On the **Display** tab, specify how the data should be interpreted (**Formatted Fields**, **Hexadecimal Fields**, or **Unformatted Records**), the display type (that is, the view mode: **Grid View**, **Tree View**, or **Formatted Text**), how to display strings (**Compact Blanks**, **Enclose in Quotes**, **Literal**, or **Substitute Mid Dot for Blanks**), and the font.
- On the **Fields** tab, specify the fields you want to display.

When you click **OK** in the **View Data Options** dialog, the **View Data** window appears:



	transaction_id	invoice_no	transaction_date	store_no	customer_id	product_cd	quantity	transaction_amt	salesperson
1	650001	250449	20001231	1179	1600082820	DXB0852	31.0	····28.76	THOMAS·LOCHER
2	650002	250449	20001231	1179	1600082820	CL04018	34.0	····16.55	THOMAS·LOCHER
3	650003	250449	20001231	1179	1600082820	0006688	28.0	····20.06	THOMAS·LOCHER
4	650004	250449	20001231	1179	1600082820	CVD4888	71.0	····21.98	THOMAS·LOCHER
5	650005	250449	20001231	1179	1600082820	FKJ5993	3.0	····17.26	THOMAS·LOCHER
6	650006	250449	20001231	1179	1600082820	MTH0140	5.0	····24.07	THOMAS·LOCHER
7	650007	250449	20001231	1179	1600082820	FKJ5993	3.0	····18.52	THOMAS·LOCHER
8	650008	250450	20001231	1114	1600057128	HJF6755	23.0	····22.07	RICHARD·HELMINIAK
9	650009	250450	20001231	1114	1600057128	DMM7303	34.0	····13.34	RICHARD·HELMINIAK
10	650010	250450	20001231	1114	1600057128	QEQ4789	8.0	····36.63	RICHARD·HELMINIAK
11	650011	250450	20001231	1114	1600057128	BQR2724	19.0	····19.41	RICHARD·HELMINIAK
12	650012	250450	20001231	1114	1600057128	CCN2562	1.0	····10.63	RICHARD·HELMINIAK
13	650013	250451	20001231	1262	1600070547	YQT7275	27.0	····25.27	GERALD·ESCOBEDO
14	650014	250451	20001231	1262	1600070547	MJA8414	12.0	····12.89	GERALD·ESCOBEDO
15	650015	250451	20001231	1262	1600070547	FKJ5993	3.0	····11.39	GERALD·ESCOBEDO

Grid view displays records in a spreadsheet-like format and is most appropriate for viewing records with simple record formats. In grid view, you can sort the records by any field by double-clicking the appropriate column heading. Double-clicking a second time reverses the sort order.

Tree view exchanges (pivots) the rows and columns. You can use tree view to collapse complex record formats, making them easier to see.

In both grid and tree views, you can:

- Adjust column widths by clicking and dragging the bar between columns, or by choosing **View > Resize Visible Columns** and then one of the submenu options.
- Specify how many additional records to view by typing a number into the **More Records** box and clicking **Go**.

PRACTICAL NOTE

Displaying the data type in View Data

In the **View Data** window, you can quickly display the data type and length of any field by hovering the mouse over the header of any column (in grid view) or over any field name (in tree view).

PRACTICAL NOTE

Switching between grid view, tree view, and formatted text

By default, **View Data** displays records in grid view. However, if you toggle to or from a different view for a given dataset, the most recent view setting is saved and becomes the default for that dataset until you change it again.

To switch between grid view, tree view, and formatted text:

1. Right-click a dataset component and choose **View Data**.
2. In the **View Data Options** dialog, click **OK**.

   The **View Data** dialog displays the records in the most recently used display type (grid view, tree view, or formatted text).
3. Do one of the following to switch to another display type:
   - Click the **Display Type** button  and choose a display type.
   - On the **View** menu, choose a display type.

Activities (**View Data**)

The following activities use the datasets in the graph **View_Data.mp** to explore the view modes and other options available when you right-click a dataset and choose **View Data**. To begin, open **View_Data.mp**.

Sorting and pivoting data

1. View the first 10 records of **Stores** in grid view. Try sorting the **zipcode** column and switching between tree view and grid view.

Using a filter expression

You can filter the data to display only records that match an expression.

▶ **To display only records that match an expression:**

1. Right-click a dataset component and choose **View Data**.
2. Enter an expression in the **Filter Expression** box on the **Records** tab of the **View Data Options** dialog, or click **Edit** to open the **Expression Editor** (described in detail in Chapter 5) and enter it there.
3. In the **Expression Editor**, click **Validate** to check the syntax.

2. View only transactions with amounts greater than $20 from the **Transactions** dataset. Use this expression:

```
transaction_amt > 20
```

Choosing which fields to view

You can limit the display to fields you specify.

▶ **To choose which fields to view:**

1. Right-click a dataset component, choose **View Data**, and click **OK** to display the data in grid view.
2. In the **View Data** window, choose **File > Options** and click the **Fields** tab.
3. Select or clear the checkboxes containing the appropriate field names, and click **OK**.

3. View only the **customer_id** and **zipcode** of the **Customers** dataset.

Activities (**View Data Unformatted**)

The following activities also use the datasets in the graph **View_Data.mp**. These activities explore the view modes and other options available when you right-click a dataset and choose **View Data Unformatted**. To begin, open **View_Data.mp** if it is not already open.

Unformatted display

The **Unformatted** display mode shows the raw data, allowing you to examine delimiter characters and other hidden characters, such as lengths for length-prefixed strings and flags associated with hidden NULL flags (described in detail in Chapter 10). To see this display, right-click a dataset component and choose **View Data Unformatted**.

4. View the first 10 records of the **Customers** dataset as unformatted text.

Formatted text

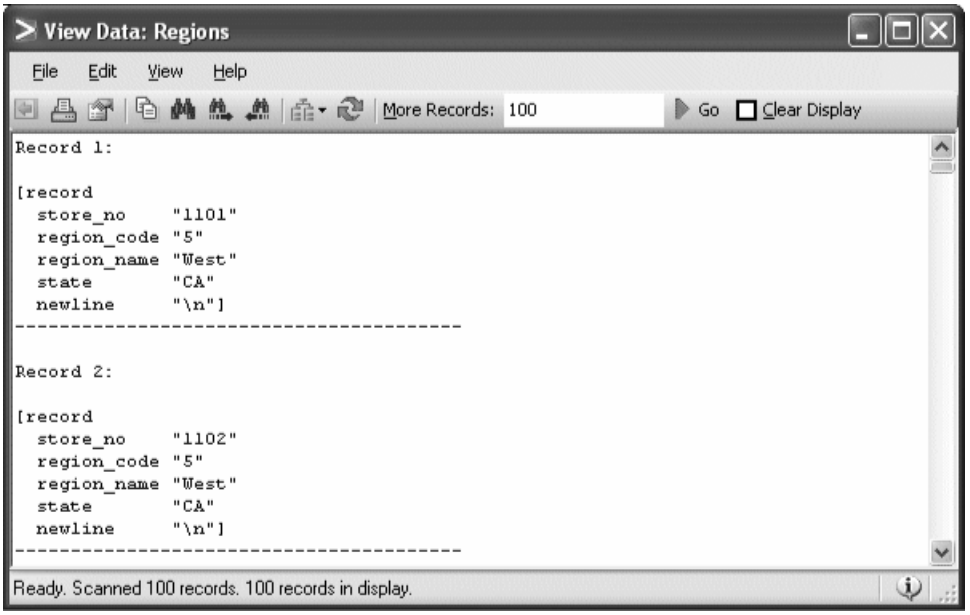
The **Formatted Text** option displays records textually, interspersing field names and field values. This display type can sometimes be a better way than grid view to look at complicated records containing subrecords and vectors. Values of string fields are displayed in quotes, so you can see all characters in the field, including leading and trailing spaces. (Note, however, that you can also see leading and trailing spaces in string fields by using **View Data**. To do this, right-click a dataset, choose **View Data**, and then, on the **Display** tab of the **View Data Options** dialog, select **Enclose in Quotes** in the **String Display** drop-down list.)

In both grid view and formatted text display, special characters (such as newlines) and certain ambiguous characters (such as double quotes) are displayed with an escape character (****). Special characters are described in more detail on page 73.

▶ **To view data as formatted text:**

1. Right-click a dataset component and choose **View Data**.
2. In the **Display Type** drop-down list on the **Display** tab of the **View Data Options** dialog, choose **Formatted Text** and click **OK**.

   Following is a **Formatted Text** excerpt from the **Regions** dataset:



5. View records 11 through 18 of **Customers** as formatted text.

Hexadecimal display

The **Hexadecimal** option can be useful for viewing data that might not be accurately described by the associated record format.

▶ **To view data as hexadecimal:**

1. Right-click a dataset component and choose **View Data Unformatted**.
2. In the **Display As** drop-down list in the **View Data** dialog, choose **Hexadecimal** and click **OK**.

6. View the first 10 records of the **Products** dataset as hexadecimal. Note how the field **whs_price**, near the end of each record, is in the EBCDIC character set.

Activity: Counting records

View Data is useful for inspecting in detail the contents of a limited number of records. However, if you want other basic information about the data — for example, how many records are in a dataset — **View Data** is generally not the best tool to use, especially with extremely large datasets. A quick way to count the records in a file is to create a very simple graph by connecting the dataset to a TRASH component (from the **Miscellaneous** folder).

Using a TRASH component to count records

7. Determine how many records are in the **Transactions** file. Build a graph that sends the **Transactions** data to TRASH. Run the graph and note the record count shown on the flow. This is the number of records in the **SmallData** sample dataset. Now choose **Project > Edit Sandbox > Parameters**, change the **TEST_FLAG** parameter to **LargeData**, save the sandbox parameters, close the editor, and run the graph again. Note the number of records in the full dataset. Remember to change **TEST_FLAG** back to **SmallData** before running the next set of graphs.

Multifiles

A *multifile* is a parallel dataset, and the data files that make up the multifile are called *partitions*. To view the records in a specific partition, you can specify a partition number.

8. View the first several records in each partition of the **Transactions** multifile in grid view.

5

Selecting by a criterion: FILTER BY EXPRESSION

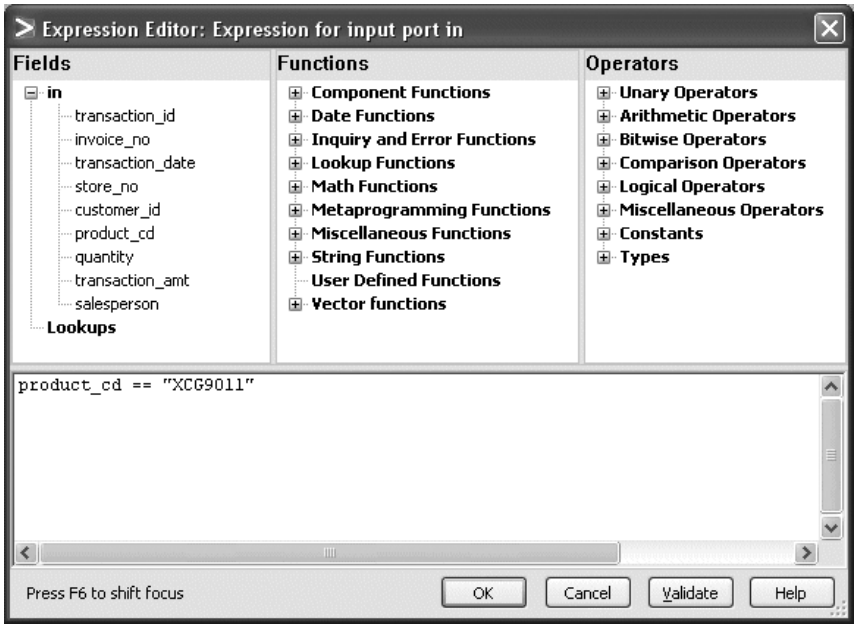
The FILTER BY EXPRESSION component (in the **Transform** category of the **Component Organizer**) performs a very basic (and very useful) function: it separates records into two groups based on a selection criterion. This is a common step in almost all processes you will implement. Furthermore, as you develop graphs or analyze data, you will find that the FILTER BY EXPRESSION component helps you track down a specific record or set of records for further investigation.

Creating a filter expression

▶ **To create a filter expression:**

1. Open a FILTER BY EXPRESSION component.
2. On the **Parameters** tab, select the parameter **select_expr** and click **Edit**.

   This opens the **Expression Editor**, which enables you to build expressions using input fields, built-in functions, and operators.

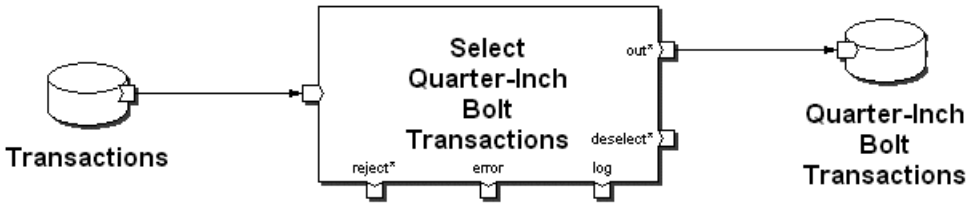


3. To enter fields, functions, or operators in the text window, double-click them in their respective panes or simply type them.

To learn more about a specific built-in function, right-click the function and choose **Help**.

If **select_expr** evaluates to a nonzero result for a record, that record is selected and sent to the **out** port; otherwise, the record goes to the **deselect** port. If there is no flow connected to the **deselect** port, the record is discarded. Note that the FILTER BY EXPRESSION component must always have something attached to its **out** port, even if it is simply a TRASH component that discards the records.

Consider Activity 1 on page 41. The suggested solution is to use a **select_expr** with a simple comparison, **product_cd == "XCG9011"**. This expression is true if **product_cd** is equal to the string **XCG9011** and false for all other values of **product_cd**. Now suppose you want to answer the complementary question, "Which transactions do not have **product_cd XCG9011**?" This could also be answered with a simple comparison: **product_cd != "XCG9011"**. You do not need to use an expression such as **if (product_cd != "XCG9011") 1 else 0**.



Using FILTER BY EXPRESSION

PRACTICAL NOTE

Shortcut to a FILTER BY EXPRESSION's select_expr parameter

If you hold down the Shift key while double-clicking a component, the GDE goes directly to the most commonly used parameter for that component so you can set its value. Doing this with a FILTER BY EXPRESSION component opens the **Expression Editor** with the **select_expr** parameter displayed.

BEST PRACTICE

Write direct and concise expressions.

Short, simple expressions are clearer — and usually more efficient — than longer, more complex expressions.

Parallelism

Some activities in this chapter introduce parallelism: both *component parallelism*, which occurs when independent (disconnected) components run simultaneously, and *pipeline parallelism*, which occurs when connected components run simultaneously. In the latter case, a component produces output for the next component while continuing to process input, enabling a "pipeline" of components to

run simultaneously. Both kinds of parallelism are "free" in Ab Initio graphs — the software exploits them automatically, and you do not have to do anything more than connect the components. A third form of parallelism, *data parallelism*, is described more thoroughly in Chapter 16.

The REPLICATE component

Some activities in this chapter use the REPLICATE component (in the **Miscellaneous** folder), which is used to make multiple copies of a flow for separate processing. REPLICATE is useful after some processing has happened. If, instead, multiple copies of a flow are needed immediately after an INPUT FILE component, you should attach multiple flows directly to the INPUT FILE component.

BEST PRACTICE

Use multiple flows instead of REPLICATE when possible.

If you need to process an INPUT FILE in several different ways, connect multiple flows directly to the INPUT FILE rather than using a REPLICATE. When downstream components are connected directly to a file, they are able to read the data independently, which can improve performance.

Activities

Use the **Transactions** dataset in each of the following activities. Start with the sandbox parameter **TEST_FLAG** set to **SmallData** (see Chapter 2).

After running your graph for each activity, use **View Data** on the output to make sure the result is what you expected. By initially running against small datasets, you can validate the results simply by looking at them. However, since some data quality issues are rare and will not show up in small datasets, you should also run your graphs on test data that includes the "edge cases." To do this for the activities in this book, change **TEST_FLAG** to **LargeData**, run the graph using the full volume dataset, and spot-check the results again using **View Data**. If you spot an unexpected error in a large-volume run, you can add that case to the small dataset for debugging and future regression testing. Remember to set **TEST_FLAG** back to **SmallData** before building the next graph.

To begin each of these activities, copy the needed dataset from the **Datasets.mp** graph and paste it into a new graph. Save the new graph as **Filter_#.mp**, where **#** is the activity number. It is a good idea to give your output files unique names (for example, **mfile:$AI_MFS/filter_out_#.dat**, where **#** is the activity number).

Identifying records with a specific value

The FILTER BY EXPRESSION component is frequently used to identify records that contain a specific value for a field. These might be the only records that are required for further processing or to answer a specific question. The desired value is identified in the **select_expr** by means of the equality (**==**) comparison operator. Note that in an expression parameter like **select_expr**, fields are referenced directly by field name and not by **in.***field_name* as in transform functions.

1. Find all transactions where a product with product code **XCG9011** was purchased.

Identifying records in a range of values

Another common task is to find records in a range of values specified by an inequality. The **greater than** (**>**) and **less than** (**<**) comparison operators, and their counterparts that allow equality (**<=** and **>=**), are used for this purpose.

2. Find all transactions where more than 20 units of a given product were purchased. Attach a TRASH component to the **deselect** port to see the number of records not satisfying the selection criterion.

Filtering the same dataset in different ways simultaneously

In a single graph, you can filter the same dataset several different ways by simply attaching multiple FILTER BY EXPRESSION components to the same INPUT FILE. The different filter processes run simultaneously; this is called *component parallelism*. (For more details on parallelism, see Chapter 16.)

3. Find all transactions where more than 20 units of a given product were purchased, and store the results in one file. At the same time, find all transactions where fewer than five units were purchased, and store the results in a second file.

Using a REPLICATE and applying another filter

The REPLICATE component is useful when you want to make multiple copies of a flow for separate processing. You might want to do this after a FILTER BY EXPRESSION component to capture the selected records in a file (or to process them further) before going on to apply a second filter to find a subset of the selected records. Both FILTER BY EXPRESSION components and the REPLICATE component will be running at the same time because they both pass records to the next component before they finish reading their input. This is called *pipeline parallelism*.

4. Find all transactions where more than 20 units of a given product were purchased, and store the results in one file. Then find all such transactions that were made in store **1224**, and store the results in a second file. Use a REPLICATE component to copy the flow after the first FILTER BY EXPRESSION component. (How would you solve the same problem using an INTERMEDIATE FILE component? Note that doing so would "break" pipeline parallelism at the INTERMEDIATE FILE component.) Be sure to test your application with **TEST_FLAG** set to **LargeData**.

Using a filter for data validation

The FILTER BY EXPRESSION component can also be used for data validation. It is often necessary to identify records with errors or problems for separate processing. You can use the built-in function **is_valid** on individual fields to identify which records contain valid values. The **is_valid** function tests whether a given value is valid for the data type. (Exactly what to do with bad data depends on business requirements and is beyond the scope of this book.)

5. Put all transactions with a valid transaction date in one file and all other records in another file.

Using the **is_blank** function

The FILTER BY EXPRESSION component can also be used to work with data that is not valid in the computational sense but that may still have meaning in a business sense. A good example is a blank date, which is often used to represent "no date set." Having no date set might be entirely valid for the business — for example, as the value of a field **closed_date** for an account that is still open. The built-in function **is_blank** can be used to identify string, decimal, or date fields that consist entirely of spaces.

6. Find all transactions with a blank transaction date.

Using a compound expression to filter for more than one property

You can check records for more than one property in a single FILTER BY EXPRESSION component by building a compound expression using the logical operators **and** (which can also be written **&&**) and **or** (which can also be written **||**).

For example, say you want to collect records in which the field **amount** is either less than **10** or greater than **100**. You could write:

```
amount < 10 or amount > 100
```

BEST PRACTICE

Make complicated expressions easier to read.

Use parentheses and whitespace to break up complicated expressions. This will not only make your original intent clear; it will also prevent unintended changes to the meaning of the expression in future development. (Neither parentheses nor whitespace affects the complexity or performance of expressions.)

If you later need to add an overall constraint that collects only records in which **kind** is equal to **"purchase"**, you cannot simply append the new constraint to the expression, as in:

```
amount < 10 or amount > 100 and kind == "purchase"
```

Because the precedence of the **and** operator is greater than that of the **or** operator, this expression would yield records for which either **amount** is less than **10**, or both **amount** is greater than **100** and **kind** is equal to **"purchase"**.

Instead, you should write:

```
(amount < 10 or amount > 100) and kind == "purchase"
```

Or to make the expression even clearer:

```
((amount < 10) or (amount > 100)) and (kind == "purchase")
```

7. Find transactions at stores **1100** through **1120** in which quantities of more than **10** but fewer than **100** were purchased (with **TEST_FLAG** set to **LargeData**).

Using the **next_in_sequence** function to count records in a serial dataset

The built-in function **next_in_sequence** (under **Miscellaneous Functions** in the **Functions** pane of the **Expression Editor**) can be used in combination with the FILTER BY EXPRESSION component to count records as they pass through the component. This function returns a number that is increased by 1 *each* time it is called, starting from the value **1**. If you are using this function to count records, you should call **next_in_sequence** only once per record, regardless of where it is used.

8. Create a sample dataset containing the first five transactions from the file **transactions.dat**.

Using the **next_in_sequence** function to count records in each partition of a multifile

The built-in function **next_in_sequence** (under **Miscellaneous Functions** in the **Functions** pane of the **Expression Editor**) can be used in combination with the FILTER BY EXPRESSION component to count records as they pass through the component. This function returns a

number that is increased by 1 *each* time it is called, starting from the value **1**. If you are using this function to count records, you should call **next_in_sequence** only once per record, regardless of where it is used. Note that each partition has its own separate sequence. (You could also do this with a more advanced component, LEADING RECORDS, but that is beyond the scope of this book.)

9. Create a sample dataset containing the first five transactions from each partition of the multifile **transactions.dat**.

SOLUTIONS

The solution graphs are named **solutions/Filter_#.mp**, where **#** is the activity number.

6

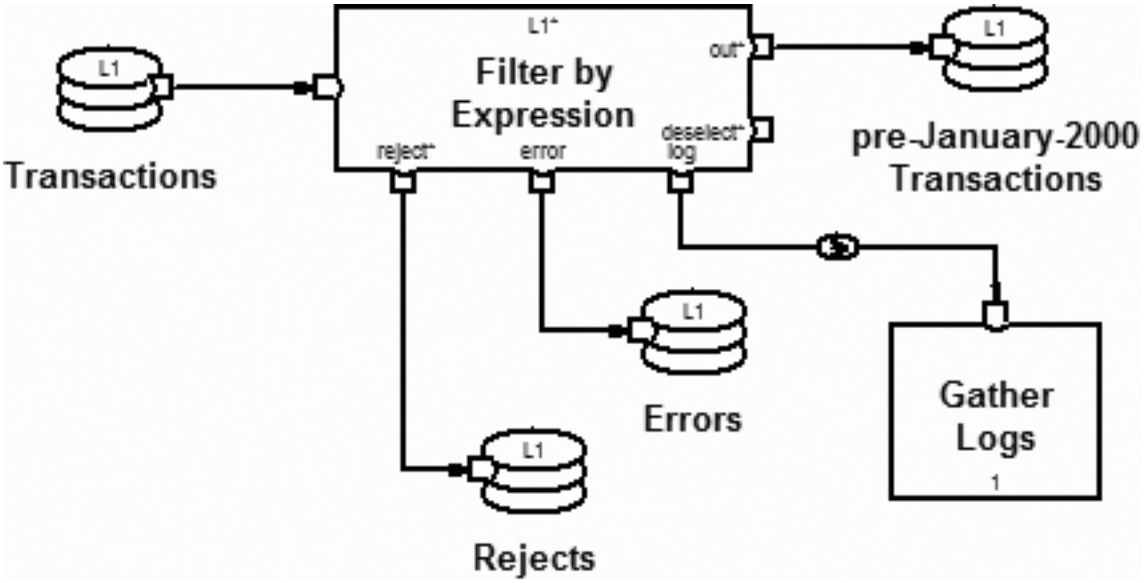
Verifying data quality: The **reject**, **error**, and **log** ports

Business rules can be difficult or impossible to apply if data is dirty or ambiguous. Suppose a business rule requires checking whether one date is later than another. Does 20-JUM-1998 come before or after 10-JUL-1998? There is no way to tell whether JUM was intended to represent June or July, so the comparison cannot be made and the business rule simply cannot be applied. Ab Initio software offers a simple way to capture such suspect data — while processing continues — by using the **reject** port available on many components. Real-world data, of course, will probably require more elaborate validity checking than is described in this chapter.

When an expression, rule, statement, or other value in a component parameter or transform cannot be evaluated sensibly for a record, that record is sent to the **reject** port. On multiple-input components (such as JOIN), the rejection of any single input record results in the rejection of *all* the

other current input records. A corresponding error message containing an explanation of the cause of the rejection is sent to the **error** port. Logging information, including information about rejections, is sent to the **log** port, where it can optionally be captured with a GATHER LOGS component (in the **Miscellaneous** folder of the **Component Organizer**).

These three ports are visible on the bottom edge of most components when you set the **Show Optional Ports** toggle button on the toolbar, or choose **View > Optional Ports**. To capture rejected records, attach an output file to the **reject** port and set the **reject-threshold** parameter inside the component to specify how many rejected records are allowed before the component aborts. The default value for **reject-threshold** is **Abort on first reject**. If the component rejects a record for any reason, the component indicator turns red and the graph halts. The other extreme, **Never abort**, is used when rejection of records is expected or intended. In this case, all rejected records are sent to the **reject** port and the graph does not halt. This can be especially useful in data validation, when you want to remove records with invalid fields from the main data flow, and in debugging, when you want to inspect the records that are causing a component to fail. You can also set the abort to occur after a certain number or proportion of rejected records.



FILTER BY EXPRESSION using the optional ports

The **force_error** function

You can use the built-in function **force_error** to direct records to the **reject** port even when there would otherwise be no rejection (that is, to reject records because something is invalid for a business reason rather than a data-computational reason). For example, suppose a record contains a gender field that can take the value **F** or **M**. Records with any other character are invalid in a business sense, but computationally there is nothing wrong with putting another character, such as **D**, in the field.

In certain special circumstances, you can also use **force_error** to treat the **reject** port as an additional data flow path leaving the component. When using **force_error** to direct valid records to the **reject** port for separate processing, you must remember that *invalid* records will also be sent there. So when using **force_error** for this purpose, set **reject-threshold** to **Never Abort**.

Log records

Log records have a standard record format that is already attached to the **log** port of a component. In addition to rejects, you can capture several kinds of log output, some of which are dependent on the type of component. To enable logging in a component, set its **logging** parameter to **True** on the **Parameters** tab and then choose the frequency of logging for each type of log record you want to capture. The default frequency is empty, meaning not to log. Non-empty values produce a log record for every specified number of records.

BEST PRACTICE

Log only information that will be regularly examined.

Often developers (or management or operations staff) believe they should log *all* errors. Doing so in production graphs is inadvisable because of the sheer volume of logging information that can be generated by a complex set of applications. However, logging all errors during development may be useful in some situations.

You can send log records to an OUTPUT FILE or other component. The GATHER LOGS component (in the **Miscellaneous** category of the **Component Organizer**) captures log output from one or more components and writes it to a file. In addition, it provides log records that capture the start and end time of the phase it is a part of. Note that the file written by GATHER LOGS is not subject to the usual recovery mechanisms; in particular, if the job fails, it is not rolled back, so the logs are available for inspection even if other output files have been rolled back to their previous state.

Experience shows that unless processes are in place for examining logs (or at the very least, deleting them over time), the logs will never be looked at and might come to fill the filesystem, eventually leading to job failure due to lack of disk space.

Activities

Save these activities as **DQ_#.mp**, where **#** is the activity number. First build and run each graph with the sandbox parameter **TEST_FLAG** set to **SmallData**. When you are satisfied that the graph runs correctly, change **TEST_FLAG** to **LargeData** and rerun the graph against the full data volume. Set **TEST_FLAG** back to **SmallData** before proceeding to the next activity.

Connecting the **reject**, **error**, and **log** ports to OUTPUT FILE components

The **reject**, **error**, and **log** ports can be connected to an OUTPUT FILE component or any other component for further processing. For these activities, you should connect the **reject** and **error** ports to OUTPUT FILE components and write the output files to the **$AI_MFS** multidirectory. The log file produced by the GATHER LOGS component should be written to the **$AI_SERIAL** directory, because the GATHER LOGS component always runs serially. To capture all rejects from a component while processing continues, you must set the **reject-threshold** parameter to **Never abort**. (If **reject-threshold** is set to its default, **Abort on first reject**, the graph will stop when the first error is encountered.)

1. Find all transactions made before January 1, 2000. Capture records containing invalid dates in a rejects file for data quality inspection. View the rejects file to see why those records were rejected. Note that with **TEST_FLAG** set to **SmallData**, you may not encounter many — or even any — data quality issues. Rerun with **TEST_FLAG** set to **LargeData**.
2. Repeat the previous activity, capturing the error messages and gathering the log messages. Set the **log_reject** parameter on the FILTER BY EXPRESSION component to **1** to log every error message. Inspect the error messages and the log file to see how rejections are described. To view the log file, right-click the GATHER LOGS component and choose **View Log**; it functions the same way as **View Data** on datasets.

Using **force_error**

As mentioned previously, the **reject** port is not just for bad data or for statements and rules that cannot be evaluated. Sometimes you will want to reject a record for other reasons. There might be a data quality issue that is not automatically detected, or you might just need finer control over how records are routed through the graph. You can use the built-in **force_error** function to reject the current record. It requires a single argument, a string (or a string expression that yields a string), which should contain a relevant message. This message is written to the **error** port of the component. You can use the **force_error** function in a FILTER BY EXPRESSION component by specifying a conditional in **select_expr**. The syntax for a conditional expression is:

```
if (condition) expr1 else expr2
```

The **else** part of a conditional expression is optional. To use **force_error** in a **select_expr**, you might write:

```
if (condition) force_error("This is bad!") else expr2
```

where the *condition* detects the records to reject, and *expr2* is the selection criterion to apply to records that are not rejected. (Note: Records for which *expr2* cannot be evaluated will also be rejected; in general you cannot assume that the output of the **reject** port consists only of records satisfying the condition. If you require this, you should be filtering on this condition in a separate filter.)

3. With **TEST_FLAG** set to **LargeData**, separate transactions into two files, those having transaction amounts less than $100 and those greater than or equal to $100. Suppose a business rule requires transactions larger than $400 to be rejected as suspect. Use **force_error** to reject these transactions into a third file for further validation.

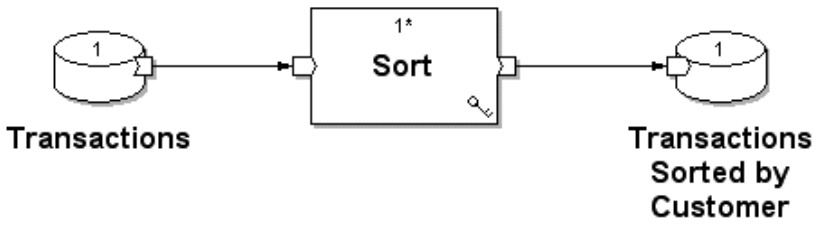
SOLUTIONS

The solution graphs are named **solutions/DQ_#.mp**, where **#** is the activity number.

7

Ordering data: SORT

Sorting is one of the most fundamental operations in data processing. In many cases, sorted data is much more efficient to work with than unsorted data. Even outside the data processing world, having sorted information helps: it would be quite a tedious task to look up someone's phone number if the phone book were not sorted alphabetically!



Sorting the Transactions dataset by customer_id

PRACTICAL NOTE

Shortcut to a SORT's key parameter

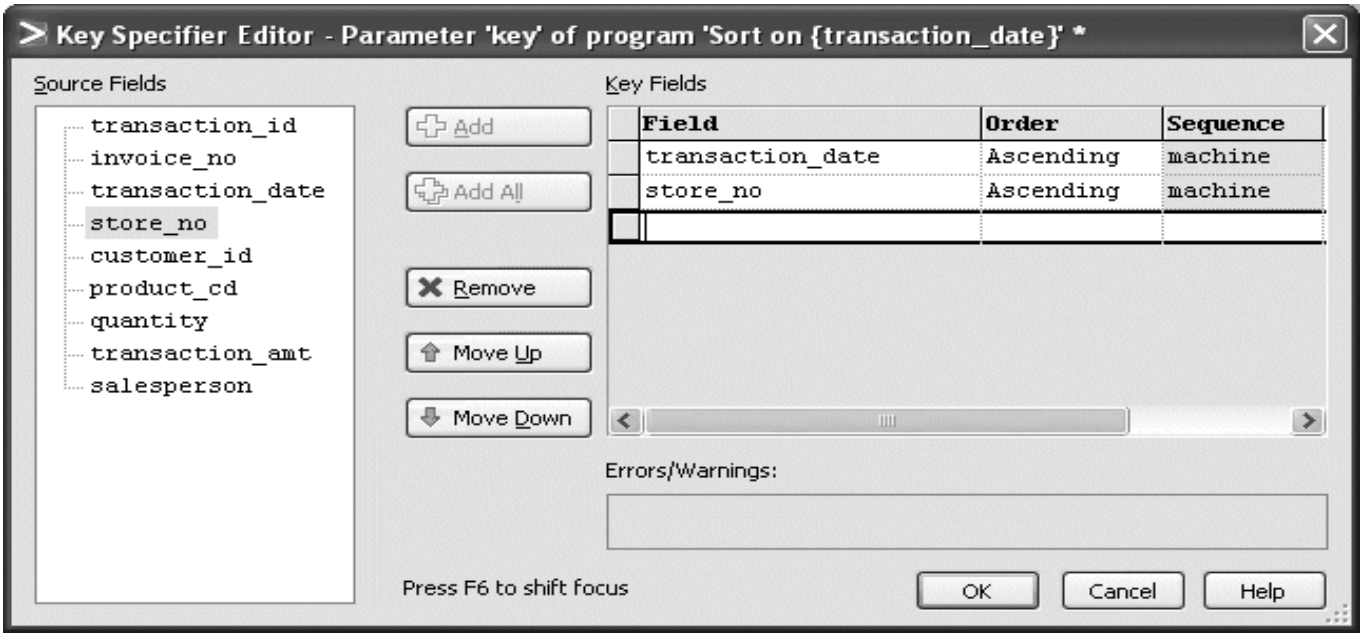
If you hold down the Shift key while double-clicking a component, the GDE goes directly to the most commonly used parameter for that component so you can set its value. Doing this with a SORT component opens the **Key Specifier Editor** with the **key** parameter displayed.

Specifying the key for a sort

To do a sort, you must specify a key on which to order the data.

▶ **To specify the key:**

1. Open the **Key Specifier Editor**.



The Key Specifier Editor

2. In the **Key Specifier Editor**, select the key field(s) on which you want to order the data: either select the field(s) and click **Add**, or simply double-click the field(s).
3. In the **Order** column, specify **Ascending** or **Descending**.
4. In the **Sequence** column, specify the type of sequence to use for arranging characters.

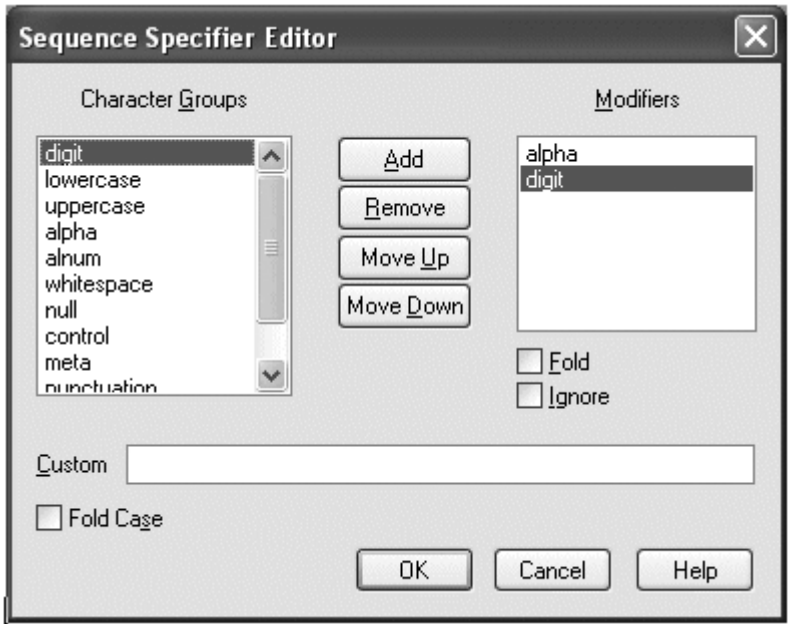
   For string fields, the default sequence (**machine**) sorts by using the raw bytes in the specified key field in numeric order as they are stored in memory. This is the fastest option. For Unix and Windows machines, this is ASCII (ISO-8859-1), which orders uppercase letters before lowercase ones. On a mainframe, this is EBCDIC, which orders

lowercase letters before uppercase ones. See the *Data Manipulation Language Reference* for more information about sequences. If you want an ordering other than **machine**, you can use a different sequence specifier.

▶ **To specify a custom sorting sequence:**

1. After specifying a key as described above, select **custom** in the drop-down list for the key field's **Sequence** column.
2. Click the pencil icon:

   The **Sequence Specifier Editor** appears:



3. In the **Character Groups** box, select the first group by which to sort the field, and either double-click it or click **Add** to move it to the **Modifiers** box.

4. Repeat Step **3** to specify additional modifiers as needed.
5. If you want to change the list, select a modifier and click **Remove**, **Move Up**, or **Move Down** as necessary.
6. Click **OK** twice to dismiss the **Sequence Specifier Editor** and the **Key Specifier Editor**.

Performance considerations and the **max-core** parameter

Sometimes it is necessary to sort records so that you can apply a set of business rules. In other cases, sorted final output might be a business requirement. Sorting a large dataset can be expensive, but the subsequent benefit in processing the data usually makes up for the cost. Many components (such as ROLLUP and JOIN) can operate on either sorted or unsorted data. If you need to do only one such operation and the volume of data is relatively small, you might find little to no benefit in sorting the data compared to using the **In-memory** option on the component. However, if a series of operations can be done using the same sort key, or if the volume of data is very large (so large that ROLLUP and JOIN components cannot do their processing in memory), sorting the data first is worth the expense.

The **max-core** parameter is the most important control you have over the performance of SORT: it controls how much memory will be made available to each instance of the SORT component. When there is more data to be sorted than will fit within the specified **max-core** memory limit, SORT works in units of approximately one-third that size, sorting the incoming data one chunk at a time and writing each sorted chunk to a temporary file. After all the data has been read, SORT does a final merge of all the temporary files to produce the sorted output.

Therefore, **max-core** controls not only the size of each chunk to be sorted, but also the number of temporary files that SORT will create. In the rare cases when memory is extremely tight, you can reduce the value of **max-core** from its default, **100 MB**. Setting **max-core** greater than the default provides little or no performance benefit for most applications. However, some operating systems have stringent limits on the number of temporary files. When you are sorting large datasets, your primary concern will generally be the number of temporary files, and you should increase **max-core**

as needed to reduce that number, setting it such that fewer than 1000 temporary files will be created on any given computer. To get a good approximation of the number of temporary files that will be needed, divide the size of the input data by one-third of **max-core**.

For example, with 1 GB (1000 MB) of input data and a **max-core** of 10 MB, SORT will create 1000/(10/3) or 300 temporary files. This is well below the recommended 1000 limit on temporary files. Note that if more memory is available for the SORT, setting **max-core** to 50 MB or using the default (100 MB) will improve performance and reduce the number of temporary files to 60 or 30, respectively. Continuing with the example, if the data volume were to increase to 3 GB with **max-core** still at 10 MB, the number of temporary files would grow to 900, approaching the recommended limit.

Sorting a large file will result in the sort data being written entirely to disk before the final merge to produce the output. If a SORT is followed immediately by a checkpoint or phase break, the sorted output will be written to disk again. The amount of working disk space will be twice the data volume: one copy of the data will reside in the SORT's temporary files and another will reside in the phase break/checkpointing file. If you are thinking of placing a checkpoint or phase break near a SORT, use the CHECKPOINTED SORT component instead. This component (in the **Sort** folder of the **Component Organizer**) separates the initial sort (of the chunks of input) from the final merge and puts a checkpoint between them. The result is that only one copy of the data will be stored on disk.

Activities

Save these activities as **Sort_#.mp**, where **#** is the activity number.

Sorting in ascending numeric order

The most common use of the SORT component (in the **Sort** folder in the **Component Organizer**) is to arrange records in increasing ("ascending") order, numerically or alphabetically. For this reason, the default direction of sorting is ascending.

1. Sort the **Products** dataset (in ascending order) on **whs_price**.

Creating a custom sort order

2. Sort the **Products** dataset on **product_name** such that products whose names begin with letters precede those that begin with digits.

Sorting on more than one field

You can arrange records based on the values of more than one field. For example, if you are grouping by city and zip code, you will want the cities sorted alphabetically, with zip codes sorted numerically within each city.

3. Sort the **Stores** dataset by city and then by zip code within each city.

Sorting on dates

In addition to sorting on strings and numbers, you can also sort on dates. (Records with invalid dates will be placed before records with valid dates in ascending order, or after records with valid dates in descending order.)

4. Sort the **Transactions** dataset by transaction date from earliest to latest. Use the serial transactions data in **file:$AI_SERIAL/transactions.dat**.

Using the CHECK ORDER component to verify that data is in sorted order

To verify that a dataset is sorted in the expected order, you can use the CHECK ORDER component (in the **Validate** folder of the **Component Organizer**). This component takes a key and checks whether the records passing through it are sorted in that order. The output port of the component does not need to be attached to anything.

PRACTICAL NOTE

How the SORT component handles invalid data

The SORT component handles invalid data not by rejecting it, as other components do, but by treating invalid values differently than valid ones. Invalid dates and decimals are treated as being "less than" valid dates and decimals, and invalid values are compared among themselves as if they were strings and then sorted.

If you send unsorted data to CHECK ORDER, it will raise an error. To see what the error looks like, use the CHECK ORDER component on any dataset that is not sorted on the key you specify (for example, send the **Products** dataset — which is sorted on **{product_cd}** — to a CHECK ORDER component with its key set to **{whs_price}**).

5. Use the CHECK ORDER component to verify that the **Products** dataset is sorted on **product_cd**.

SOLUTIONS

The solution graphs are named **solutions/Sort_#.mp**, where **#** is the activity number.

8

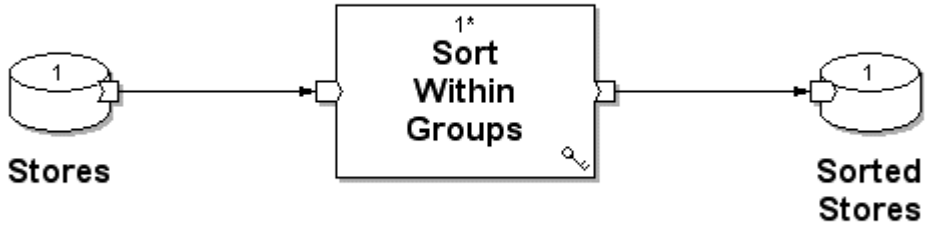
Refining the order of sorted data: SORT WITHIN GROUPS

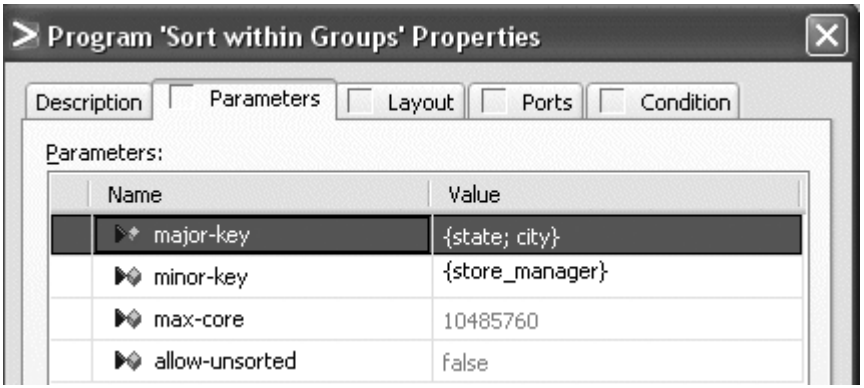
If your data is already sorted, you can use a SORT WITHIN GROUPS component to refine or modify the order in certain ways without completely re-sorting it from the beginning. For example, if your transactions dataset is already sorted by product and amount paid, you can create a dataset sorted by product, amount paid, and purchase date more efficiently by using a SORT WITHIN GROUPS than by re-sorting the full dataset.

Using the **major_key** and **minor_key** parameters

The SORT WITHIN GROUPS component (in the **Sort** category of the **Component Organizer**) refines the order of your sorted dataset by further sorting according to an order specified by its **minor_key** parameter within an order specified by its **major_key** parameter. Input records that share common values of the major key might initially occur in any order in the minor key. You can use SORT WITHIN GROUPS if the input data is sorted by the major key.

Because the number of records that share a common major key is generally very small, SORT WITHIN GROUPS does not need to spill partial results to temporary files and then merge them (as a full sort might require), so it is generally quite fast. In addition, the SORT WITHIN GROUPS component can work in pipeline fashion, raising the degree of pipeline parallelism an application can achieve.





Using SORT WITHIN GROUPS to refine the order of the Stores dataset

Activities

Save these activities as **SortGroups_#.mp**, where **#** is the activity number.

Sorting a sorted dataset by an additional key

The **Stores** dataset is sorted by store number within each state and city.

1. Sort the stores by manager within each state and city without re-sorting the full dataset. (Note that because the **store_manager** field is a single string containing both first name and last name, with first name first, a manager's first name will determine the store's position in the ordering.)

Using a REPLICATE to re-sort in a different way

Sometimes you will want to use SORT WITHIN GROUPS to get a second ordering of a dataset for a second purpose.

2. Sort the serial **Products** dataset by last modified date and arrange the products in descending order by wholesale price within each date. Use a REPLICATE component to copy the flow. Connect one flow to an output file to store the result of the sort in a file. On a second flow from the REPLICATE, use SORT WITHIN GROUPS to rearrange the products in ascending order by wholesale price within each date, and store the result in a second file.

SOLUTIONS

The solution graphs are named **solutions/SortGroups_#.mp**, where **#** is the activity number.

9

Removing duplicates: DEDUP SORTED

The DEDUP SORTED component (in the **Transform** category of the **Component Organizer**) removes duplicate records from a sorted flow of data. A *duplicate* is a record with the same key value as another. Sometimes duplicate records arise because of data quality problems; at other times they are an artifact of how a dataset was created. For example, if data containing point-of-sale purchases is merged with data containing mail-order purchases, the resulting dataset might contain some customers that are represented multiple times, one from each transaction stream. To obtain the set of unique customers, you need a dataset in which each customer is represented by a single record.

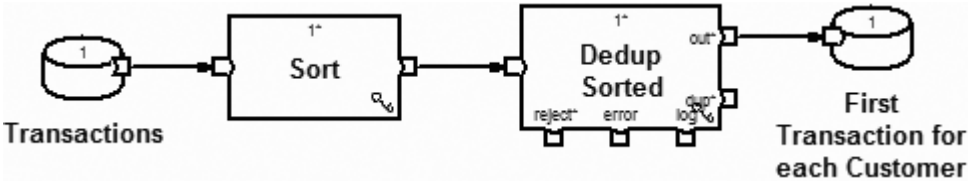
- **To remove duplicates from a dataset by using the DEDUP SORTED component:**
  1. Sort your data using the appropriate key (if it is not already sorted).
  2. Set the key for grouping the records in the DEDUP SORTED component.

3. Using the **keep** parameter, specify whether to keep the first record for each element, the last record, or only records with unique key values.

Determining which records to keep

In some situations, records sharing the same key will have identical contents and represent truly redundant information. Removing such duplicates is an easy part of data cleansing. In other situations, however, the record contents are not exactly the same and you must apply a business rule to determine which records to keep. If the record you want to keep can be brought to the beginning or the end of the group by appropriate sorting, you can use the DEDUP SORTED component to get that record.

For example, there might be multiple transaction records for a single customer, and the business rule is to keep the record with the oldest **last_modified_date**. You can easily find this record for each customer by sorting on the **customer_id** and **last_modified_date** fields and using DEDUP SORTED with the key set to **{customer_id}** and the **keep** parameter set to **first**.



The DEDUP SORTED component

Activities

Save these activities as **Dedup_#.mp**, where **#** is the activity number. For each activity, remember to filter out invalid dates.

Using SORT with DEDUP SORTED

In the **Transactions** dataset, each customer is uniquely identified by a **customer_id** field.

1. Using the serial **Transactions** file (**file:$AI_SERIAL/transactions.dat**), sort the transactions on **customer_id** and **transaction_date** (ascending). Then apply the DEDUP SORTED component to build a dataset containing only the earliest transaction record for each customer and thus the date of each customer's first purchase.

Using DEDUP SORTED's **keep** parameter

The DEDUP SORTED component allows you to specify which records to keep by modifying the **keep** parameter of the component. Your choices are to keep either the first or last record in a group, or only records with unique key values. The default value of the **keep** parameter is to take the first record from each group.

2. Build a dataset containing only the most recent transaction record for each customer and thus the date of each customer's last purchase.

Using DEDUP SORTED with a compound key

You can remove duplicates based on a compound key, just as you can sort on keys with more than one field.

3. Find the first transaction made each day in each store. Assume that successive transactions on any given day are assigned successive **transaction_id** numbers.

Using SORT with a compound key before DEDUP SORTED

Nested requirements, such as "Choose the record with the smallest amount; if multiple records have the same amount, take the one with the lowest store number," can be implemented by sorting on multiple fields: inner sort keys "break ties" of outer ones. In the following activity, the sort key is **{transaction_date; transaction_amt; store_no; customer_id}**.

4. Find the smallest transaction (in terms of amount) made each day. If multiple transactions have the same amount, take the record with the smallest store number. If multiple transactions have the same store number, take the transaction with the lowest **customer_id**.

Mixing ascending and descending sort orders

A descending sort puts the "largest" record of a group at the start, while an ascending sort puts it at the end. Because the value of DEDUP SORTED's **keep** parameter (**first** or **last**) applies to all keys used in the upstream SORT component, you may have to mix ascending and descending sort orders for the different parts of a compound key to ensure that you get the results you want.

5. Find the largest transaction (in terms of amount) made each day. If multiple transactions have the same amount, take the record with the smallest store number. If multiple transactions have the same store number, take the transaction with the largest quantity.

Using the empty key to treat all records as a single group

6. Extract the last record from the **Products** dataset.

SOLUTIONS

The solution graphs are named **solutions/Dedup_#.mp**, where **#** is the activity number.

PRACTICAL NOTE

Using the empty key

When the **key** parameter for the DEDUP SORTED component (and others) is set to the empty key: **{}**, all records in the flow are treated as a single group. This is useful for getting the first or last record — for example, when a dataset has a header or trailer record.

To create an empty key:

1. On the **Parameters** tab of the component's **Properties** dialog, select the **key** parameter.
2. In the **Value** box, enter **{}**.

10

Parsing data: Record formats and DML files

Ab Initio graphs use the Ab Initio Data Manipulation Language (DML) to specify record formats, expressions, transform functions, and keys. "DML" is often used to refer to record formats, which often reside in files with the **.dml** extension and are associated with the ports of a component.

A record format describes how data is structured and how it should be interpreted. An accurate description of record structure is a prerequisite for using and accessing fields of that structure. When creating your own record formats, you should name your fields descriptively so others will know what is in them.

PRACTICAL NOTE

About grid mode and text mode

DML types and transforms can be viewed and edited in one of two modes: grid mode or text mode. Unless you toggle the mode by clicking the **Grid/Text Views** button  on the main toolbar, the **Record Format Editor** opens in the most recently used mode. Clicking this button toggles the setting for both the **Record Format Editor** and the **Transform Editor**, whereas switching the setting from the **View** menu of either editor affects only the view/edit setting for that editor.

Using the Record Format Editor

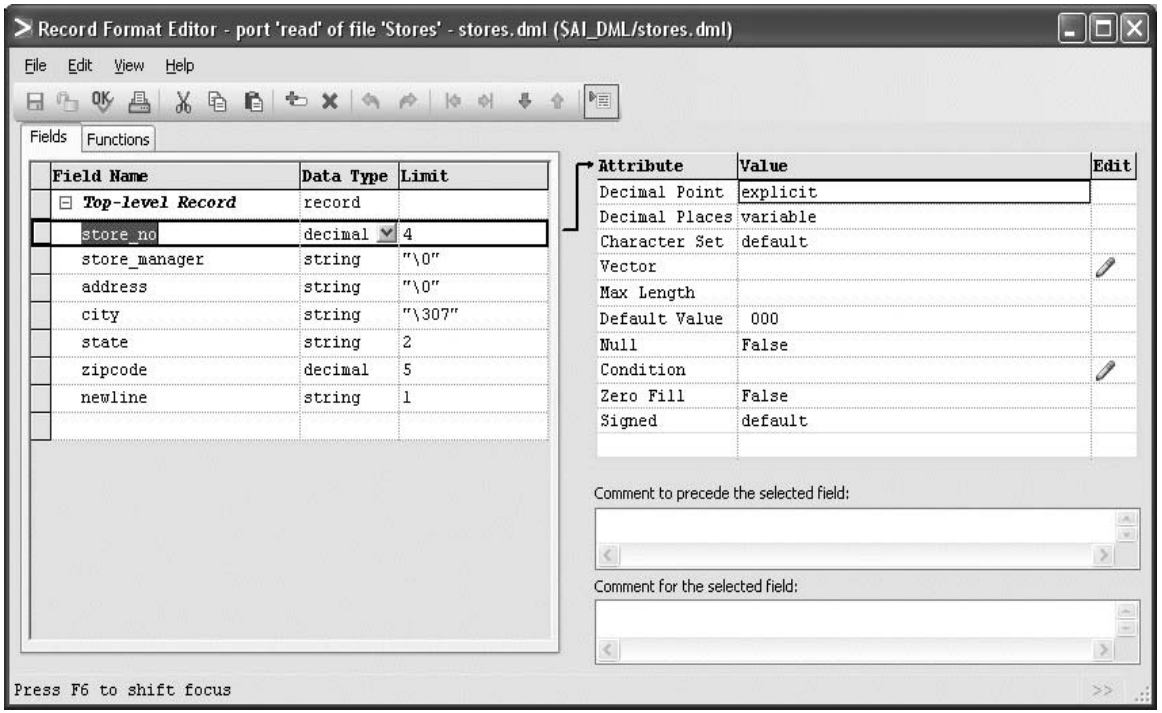
The activities in this chapter provide practice in using the **Record Format Editor** to create and modify record formats.

▶ **To open the Record Format Editor:**

- Do one of the following:
  - Double-click the appropriate port.
  - Double-click the component to display the **Properties** dialog, click the **Ports** tab, select the port whose format you want to edit, and click **New**.

▶ **To add a field to a record type:**

1. In the **Field Name** column, type the field name.
2. In the **Data Type** column, choose the field's basic data type from the drop-down list.
3. In the **Limit** column, specify the length (or delimiter) unless you are creating a date type.
4. Specify any required attributes (such as **Format** if you are creating a date type).
5. To check the syntax of the record format, choose **File > Validate** or click the **Validate** button:  .



In the activities in this chapter, you will create DML for data files that already exist. In practice, of course, you will often create DML for data that your graphs create. You can use DML in many ways to represent the same data, and in general you should strive to use the representation that is most efficient but still provides the flexibility and expressiveness you need.

Activities

The following activities consist of building record formats that describe a number of different datasets. Put all the datasets for these activities in a single graph, **RecordFormat.mp**.

For each activity, enter the record format in a new INPUT FILE component. Enter the file URL **file:$AI_SERIAL** and then browse for the file **record_format_#.dat**, where **#** is the number of the activity. After building your record format, check that it is correct by using **View Data** on the dataset. If the record format is correct, the data should look reasonable.

Fixed-length fields

Generally, fixed-length fields are processed more efficiently than variable-length fields because their size does not change from one record to the next, and Ab Initio software exploits that fact at many levels. The fewer decisions the software needs to make for each record, the faster the data can be processed. Therefore, records made up entirely of fixed-length, nonconditional, and non-nullable fields are generally processed more efficiently than records with variable-length, conditional, and nullable fields. For fields that are used as keys, additional performance advantages result from using fixed-length types.

However, there are some cases where using fixed-length fields will introduce huge inefficiencies. For example, if the length of most values of a string field is five to eight characters, but you must allow for values of up to 60 characters, it would probably be unreasonable (in terms of disk storage, memory use, and network bandwidth) to use a fixed-length field of 60 characters.

1. Create a record format consisting of three fixed-length fields: **first_name**, **cust_no**, and **rec_count**. Set their data types to **string**, **decimal**, and **integer**, and their sizes to **6**, **10**, and **4**, respectively.

Delimiters

Delimited fields can be used to store variable-length data. Using them can reduce the size of the data, at the cost of higher runtime overhead: the Co>Operating System must search the data for the delimiter.

Three very frequently used delimiters are comma (**","**), Ctrl+A (**"\001"**), and newline (**"\n"**). The comma is very often used when data is exported from software like Microsoft Excel. Ctrl+A is often used by Ab Initio database components. Newline is often used as a field or record delimiter on Unix systems.

The Ctrl+A and newline delimiters are two examples of special characters that must be entered in the **Record Format Editor** with a backslash escape character (****). Another example is double quote (**\"**). **View Data** also displays these characters with a backslash in grid view and formatted text view. For more information on escape characters and how they are used to express and display strings with special characters, see the *Data Manipulation Language Reference*.

2. Create a record format consisting of three delimited fields: **cust_no**, **fullname**, and **address**. Set their types to **decimal**, **string**, and **string**, and their delimiters to **","**, **"\001"**, and **"\n"**, respectively.

Record delimiters

You can associate a delimiter with a record as a whole instead of (or in addition to) an individual field. To set a record delimiter in the **Record Format Editor** in grid view, put the delimiting characters in the **Limit** field next to **record** at the start of the record format. (In text view, enter the delimiter in parentheses after the final **end**.)

3. Create a record format with a **"|"**-delimited ("pipe-delimited") **string** field **name** and a 10-digit **decimal** field **cust_no**, and delimit the record with a newline, **"\n"**.

BEST PRACTICE

Use punctuation fields rather than record delimiters when possible to describe the structure of records.

Restrict your use of record delimiters to situations where you need them to deal with data quality issues.

Using a record delimiter as described above is usually overkill. The previous data can also be described as a record with an additional one-character string field at the end. This is much more typical and generally more efficient as well. Record delimiters are useful primarily in situations where data quality problems might result in incomplete records in a flow.

Default field values

Given an additional "punctuation field," do you then have to write a transform rule to populate it wherever data of this format is produced? No — field default values can be used here. The field in this case should be declared as a one-character string with default value **"\n"**.

Default field values are useful in record formats other than those that use punctuation fields at the end. Anytime you want to make it possible for a record to be produced without an explicit assignment to a specific field, give that field a default value. For example, the empty string, **""**, is not a valid date, but it is often used to signify "no date set" and so can make a sensible default value.

Default values are specified in the **Attribute** pane of the **Record Format Editor**. If the **Attribute** pane is not visible, you can turn it on by choosing **View > Attributes** from the **Record Format Editor** menu bar.

4. Create a record format with default values assigned to four fields. Assign the default value **""** (empty string) to the eight-character string **firstname**, the default value **""** to the date field **end_date** with format string **"YYYYMMDD"**, the default value **0** to the four-byte integer **count**, and the default value **"\n"** to the one-character string **newline**.

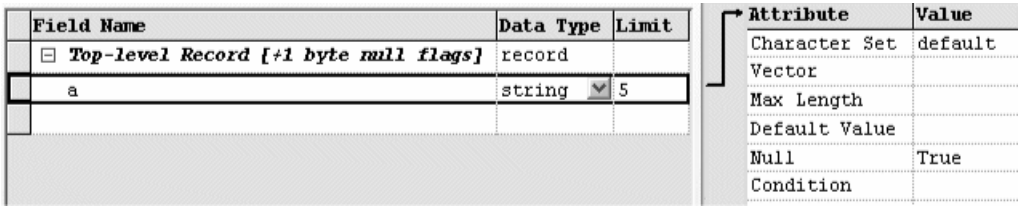
Nullable fields with defaults or hidden NULL flag representation

Nullable fields allow records to carry NULL values. If a nullable field has a default value, when that default value appears in the field, the field is treated as NULL. Nullable fields are most useful in graphs that interact with database tables that have nullable fields. Nullability is set in the **Attribute** pane in the **Record Format Editor**.

You can choose to represent NULL through the field's default value, as is most commonly done, or through a "hidden NULL flag."

To specify that a field should be interpreted as NULL when its value is equal to its default value, you need to both set the **Null** attribute to **True** and specify a default value in the **Attribute** pane. Such a field is said to have a "NULL default value."

If a nullable field has no default value, a hidden NULL flag representation will be used. This is indicated by text like **[+1 byte null flags]** next to **Top-level Record** in the **Record Format Editor**.



A field that uses a hidden flag to represent NULL

Note that the "hidden flags" are actually part of the data when viewed outside Ab Initio software or when viewed with a different record type. If you ever find yourself wondering how "garbage" or "control characters" ended up at the beginning of each of your records, it is very likely that you are looking at data that was created with a record format that had one or more fields using a hidden NULL flag representation.

BEST PRACTICE

Use the most efficient means of representation that still provides the required flexibility and expressiveness.

- If using a fixed-length type does not introduce unacceptable overhead in terms of storage, use a fixed-length type.
- If a field will never take on the value NULL, do not make it nullable.
- If a field can be NULL and a value can be reserved to represent NULL, use a NULL default value instead of a hidden NULL flag.

5. Create a record format with a 10-digit decimal **cust_no**, a nullable **"\001"**-delimited string **fullname**, and a 10-digit decimal **trans_total** that can be a NULL field with a default value of

–1. Compare **View Data** in formatted text and in hexadecimal modes to see the hidden flags that indicate NULL. The leading byte of each record holds the hidden NULL flags. In this case, there is only one field with a hidden NULL flag, and so only one bit there is used: the lowest-order bit indicates whether the **fullname** field is NULL.

Length-prefixed strings (varstrings)

Length-prefixed strings (sometimes called *varstrings*) include length information in the data before the actual string data. Because they indicate their length up front, length-prefixed strings can be more efficient than delimited strings while still providing variable-length flexibility. A disadvantage is that the extra length information is embedded in the data. To specify a length-prefixed string type in grid view, select the pseudo-type **varstring** and put the type of the object that is used for the length in the **Limit** column. In text view, the **string** keyword is used for fixed-length and delimited strings; similarly, a length-prefixed string with a two-digit **decimal** length is specified as **string(decimal(2))**. In the unformatted mode of **View Data**, you can see the length preceding the characters in the string. Binary types like **integer(2)** are commonly used to hold lengths, as in **string(integer(2))** — but this binary data, of course, might make it difficult to inspect or edit data in ordinary text editors.

PRACTICAL NOTE

Specifying hours and minutes in datetime fields

For hours in a **datetime** field, you should specify either 24-hour time (by using **HH24** in the format string) or AM/PM (by using **AM** or **PM** in the format string). Using **HH12** or **HH** in the format string without **AM** or **PM** will lead to errors when you process data past midday. Be sure to use **MI** to specify minutes in a **datetime** format string. **HH:MM:SS** might look nice, but remember that **MM** stands for *months*, not minutes.

6. Create a dataset with a single field **firstname** of type **varstring** with length **decimal(2)**. Use **View Data** to display the strings in both formatted text and unformatted modes.

Dates and times

The **date** and **datetime** types in DML require a format string that shows the placement of the parts of the date or time and the associated punctuation.

7. Write a record format consisting of two **date** formats and one **datetime** format to hold the following representations of March 1, 1999: **"03011999"**, **"01-MAR-1999"**, and **"1999030116:03:20"** (the last of these is **4:03:20 PM** in 24-hour time on March 1, 1999). Choose your own field names.

Explicit and implicit decimal points

In record formats, decimal fields can have an explicit or implicit (or "implied") decimal point, or no decimal point at all.

An *explicit* decimal point means that the decimal point character (.) is present in the raw data and will be seen there if it is viewed. In grid view in the **Record Format Editor**, the total number of characters (including the decimal point) and the number of places after the decimal point can both be entered as part of the length (such as **10.2**). The first number in this "length" is the total length of the decimal field, including the decimal point character and the decimal places. Alternatively, this can be specified through the **Decimal Point** and **Decimal Places** attributes (setting them to **explicit** and **2**, respectively).

In contrast, an *implicit* decimal point is not present in the raw data. This is specified with a comma (**,**) in the **Limit** column in grid view, as in **10,2**. Note that the implicit decimal point is never present in the data, so it is not counted in the total length. Thus, a **decimal(10,2)** value (implicit decimal point) cannot necessarily be stored in a **decimal(10.2)** field (explicit decimal point) — there might not be enough room. In that case, you would need to use the type **decimal(11.2)** to hold the explicit-decimal representation.

You can also use explicit and implicit decimal points with delimited decimals. For example, a comma-delimited decimal with two explicit decimal places is represented by the type **decimal(",".2)**.

Data read from another platform (often a mainframe) might be in a different character set. You can specify the character set for a field in the **Attribute** pane in the **Record Format Editor**.

8. Create a record format consisting of two EBCDIC decimal fields, **dollar_amt** and **mkt_percent**, each 10 characters long — the first with an explicit decimal point with two places after the decimal and the second with an implicit decimal point and one place after the decimal. Use **View Data** to display the data in formatted, hexadecimal, and unformatted modes.

Note that the **View Data** display of **record_format_8.dat** does not show implicit decimal points. Thus, the numbers in the **mkt_percent** column in the **View Data** display do not have decimal points. However, any calculations done on that data will treat the decimal point correctly. For example, although the first value in the **mkt_percent** column is **481**, any mathematical operations done on it will treat its value as **48.1**, because its format is **decimal(10,1)**.

Subrecords

In DML, a record can contain a record: this is called a *subrecord*. In grid view in the **Record Format Editor**, you create subrecords by choosing the type **record** for a field. While the containing record is expanded in the editor, fields below it are added to the subrecord.

9. Create a record format containing a **string(10)** field **account_key**, a subrecord **phone_number** with two fields — a **decimal(3) area_code** and a **decimal(7) local_phone_number** — followed by a **string(4)** field **plan_type** and a **string(2)** field **newline** with a default value **"\r\n"** (the typical Microsoft Windows newline sequence, also known as carriage return, linefeed — or CRLF).

COBOL copybooks

You can import COBOL copybooks by opening the **Record Format Editor**, choosing **File > Import COBOL** from the menu bar, and filling in the appropriate information in the dialog that appears. Other ways to import record formats might also be available, depending on the products you have purchased and installed. You might, for example, have the option of selecting **Import ERwin V4** to create a record format from an ERwin Version 4 **er1** file.

10. **$AI_DML/smp.cpy** is a COBOL copybook. Convert it to a DML file. Compare the COBOL copybook with the corresponding record format.

SOLUTIONS

The solution graph containing all the record formats is **solutions/RecordFormat.mp**.

11

Automatic type conversion: REFORMAT without a transform function

Unlike components that require transform functions you define, the REFORMAT component (in the **Transform** folder of the **Component Organizer**) can operate with or without a transform function. When a transform function is not supplied, REFORMAT uses DML's value assignment rules to produce an output record for each input record.

Value assignment rules

Following are some examples of how the REFORMAT component can automatically convert values from one type to another:

- A fixed-length **string(5)** value can be automatically converted to a delimited **string(",")** value.
- A binary **integer(4)** value can be automatically converted to **decimal(8)** value.
- A **date("YYYYMMDD")** value can be automatically converted to a **date("DD-MMM-YYYY")** value.

For detailed information on value assignment rules, search for **"value assignment"** in Ab Initio Help.

The most powerful kind of automatic conversion is that between **record** types. Sometimes called *default record assignment*, this process occurs as follows, and is applied recursively to subrecords:

1. Fields of the output are matched up with like-named fields of the input.
2. Values are assigned (and converted using value assignment rules, if necessary) from input to output.
3. Output fields that have no corresponding input field are assigned their default value, if any.

Some restrictions apply to conversion of values between types. This "type safety" prevents nonsensical or ambiguous conversions, such as that between a binary integer and a string, or a string and a record. In the individual cases where there might be some sensible way to convert between such types, you must codify the conversion in a transform function.

Activities

Save these activities as **Rfmt_wo_xfr_#.mp**, where **#** is the activity number.

▶ **To create a DML file that is similar to another one:**

A very easy way to create a DML file that is closely related to an existing one is to follow these steps:

1. On the **Ports** tab of a component, click **Use another port in graph**.
2. From the **Port in graph** drop-down list, select the port carrying the DML you want.
3. Click **Embed** to make a local copy of the DML file within the component.
4. Click **Edit** to modify the new DML file.

Converting a field from EBCDIC decimal to ASCII decimal

The default character set on Unix or Microsoft Windows is ASCII; on MVS it is EBCDIC. You can set character sets explicitly through the **Character Set** attribute in the **Record Format Editor**, associating a character set with each field or with an entire subrecord by specifying the character set on the line where the field or record is defined.

1. Build a graph that uses a REFORMAT component without a transform function to change the type of the **whs_price** field of the **Products** dataset from an EBCDIC decimal to an ASCII decimal.

Using automatic date format conversion

Date formats specified in the **Record Format Editor** can take on a wide variety of forms (see the *Data Manipulation Language Reference*), and conversion between formats is part of automatic type conversion.

2. Using the serial **Transactions** dataset, build a graph that uses a REFORMAT component without a transform function to change the format of the **transaction_date** field from **YYYYMMDD** to **MMM-DD-YYYY**. Capture invalid data from the **reject** port.

Dropping unneeded fields

BEST PRACTICE

Reduce data volume as soon as possible.

In general, you should reduce the data volume as early as possible in a graph. One way to do this is to decrease the record size by dropping unneeded fields. Another way, when appropriate, is to filter the data to remove unneeded records.

Default record assignment can be used to drop fields as well as change their types.

3. Build a graph that uses a REFORMAT component without a transform function to remove the **customer_id** field from all transactions.

Specifying default values for fields

Supplying a default value for a newly added field enables you to avoid modifying transforms when record formats change (perhaps because of changing requirements).

4. Build a graph that uses a REFORMAT component without a transform function to add a new field, **discount**, to the **Products** dataset. Make it a four-character decimal with one place after an explicit decimal point, and a default value of **0**.

SOLUTIONS

The solution graphs are named **solutions/Rfmt_wo_xfr_#.mp**, where **#** is the activity number.

12

Transforming data: REFORMAT

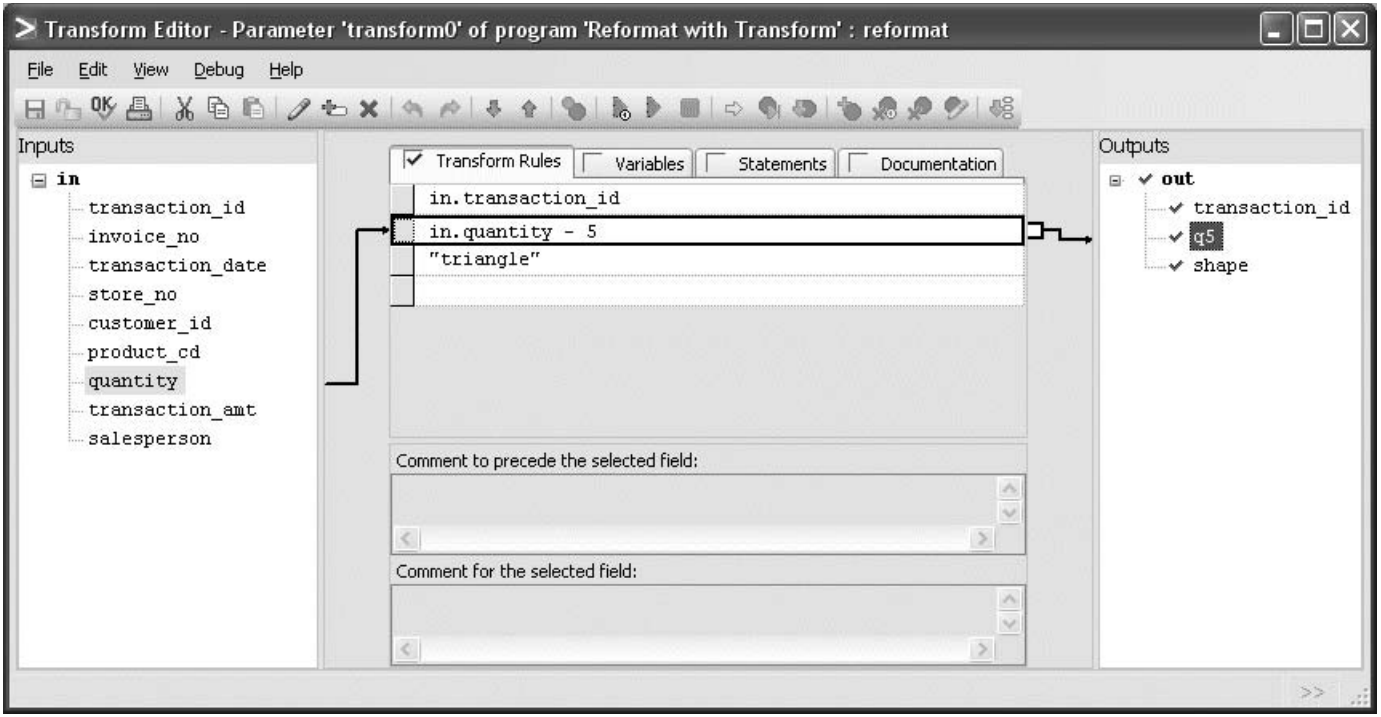
Transform functions drive nearly all data transformation and computation in Ab Initio graphs, and the REFORMAT component is the simplest component that uses them. You can use the REFORMAT component to manipulate the contents of a flow, one record at a time. By using DML expressions, you can split, remove, or rename incoming fields, or combine them to create new ones. With a single expression, you can extract the year from a date, drop an address, combine first and last names into a full name, or determine the amount owed in sales tax from a transaction amount; with a transform function, you can do all these things at once. In addition, you can validate and cleanse existing fields, deleting bad values, setting default values, standardizing field formats, or rejecting records containing invalid data. To define all these computations, you use DML expressions in the **Transform Editor**.

PRACTICAL NOTE

Shortcut to a REFORMAT's transform0 parameter

If you hold down the Shift key while double-clicking a component, the GDE goes directly to the most commonly used parameter for that component so you can set its value. Doing this with a REFORMAT component opens the **Transform Editor** with **transform0** displayed.

The **Transform Editor** specifies the transform rules that use fields in the input record to produce the fields of the output record. Right-clicking a rule and choosing **Edit Rule** from the pop-up menu opens the **Expression Editor** (described in more detail in Chapter 5). To check the syntax of the rule, click **Validate** in the **Expression Editor**. To check the syntax of an entire transform, choose **File > Validate** in the **Transform Editor**.

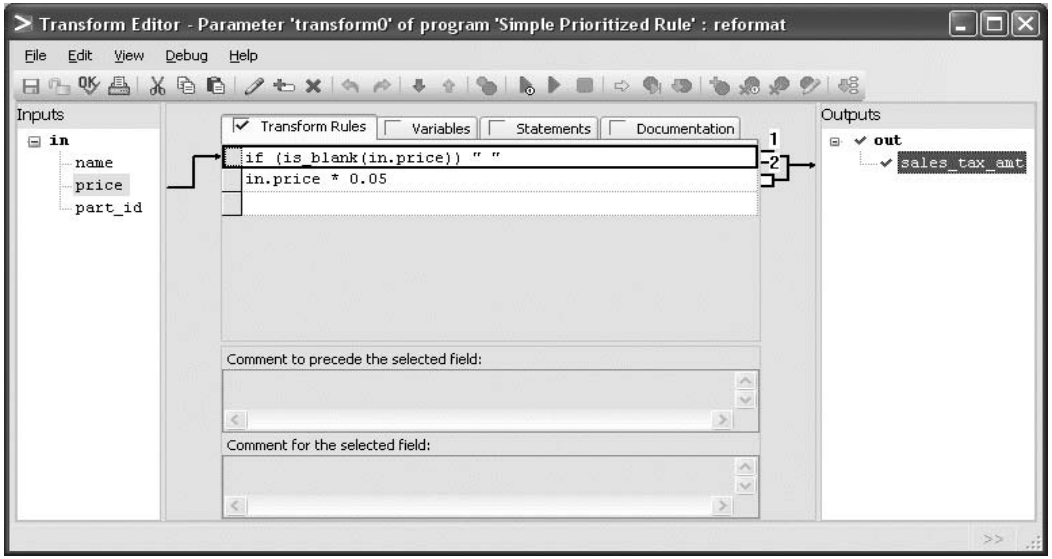


The Transform Editor is used to specify transform rules.

Prioritized rules

With *prioritized rules*, you can attach more than one rule to a single output field. The transform tries to evaluate the rules in order of priority until one of them yields a non-NULL value, which it then assigns to the output.

Consider the following business requirement: suppose you need to compute the sales tax at a rate of 5% of the price of a product. If the **price** field is blank, the resulting sales tax should be blank; otherwise, the sales tax is the price multiplied by **0.05**. This kind of requirement can be simply and directly handled in the transform of a REFORMAT component using prioritized rules. In this example, two rules will give values for the output field **sales_tax_amt**.



A prioritized rule in the Transform Editor

PRACTICAL NOTE

Viewing a field's rules and their priorities

To see which rules are associated with a field (and show their priorities), click that field in the output record in the **Transform Editor**.

In the **Transform Editor**, priorities are initially assigned to rules in the order in which they are attached to the output field. The last rule attached to the output field is always given a blank priority, which places it after all others in priority. Rule priorities before the last are indexed from **1** by default, and each rule must have a distinct priority. To change the priority of a rule in the **Transform Editor**, right-click the transform rule and choose **Set Priority** from the pop-up menu. Note that the priorities for the rules assigned to a given output field do not have to be consecutive.

The transform tries to evaluate the rules in increasing order of priority (the rule with blank priority, if any, comes last). If a rule results in NULL, the next rule is tried. Once any rule for a specific output returns something other than NULL, that value "wins" and no others are tried. If none of the rules gives a non-NULL value, the default value for the field is assigned. If there is no default value and the field cannot be NULL, an error is raised (and the input record is usually rejected).

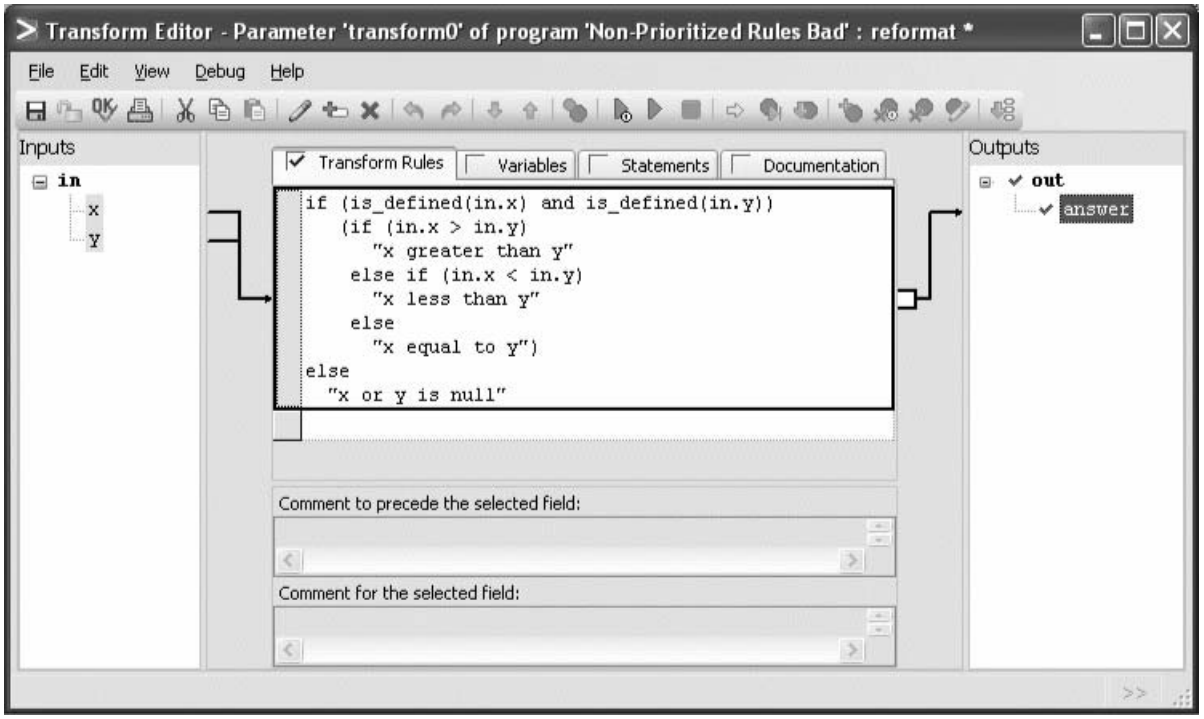
When writing transform rules, follow these guidelines:

- Write your transform rules to ensure the desired level of data quality.

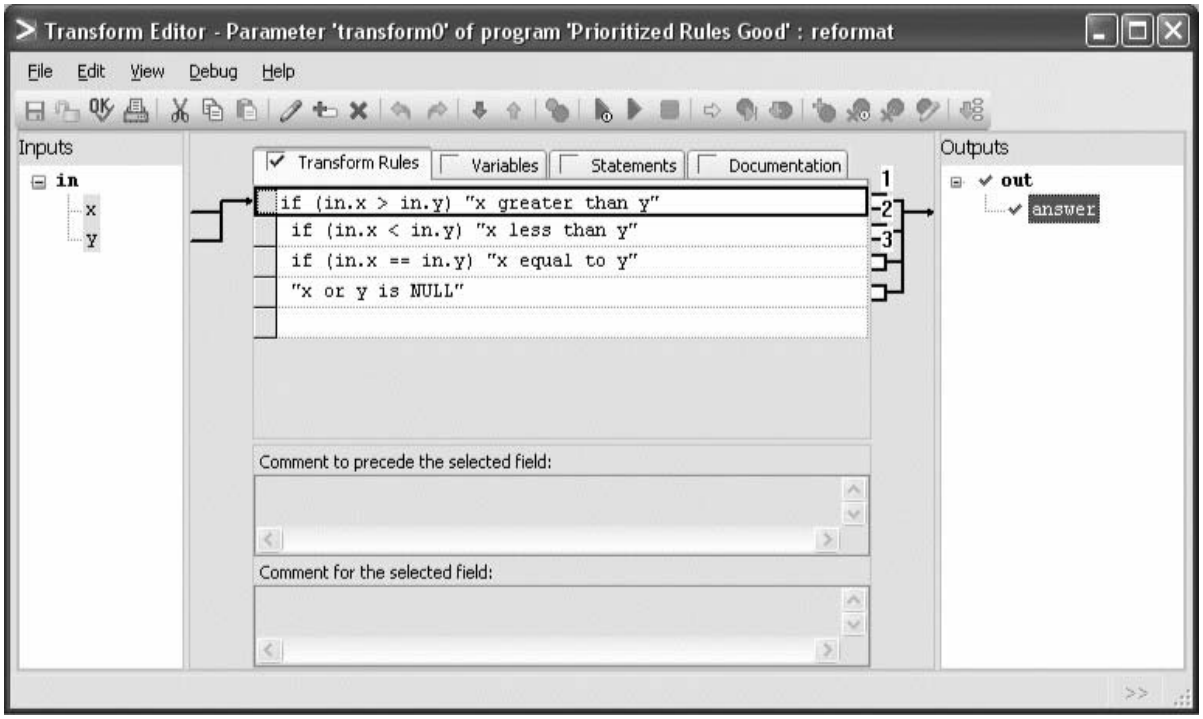
  If certain data quality issues are well known and there are well-defined things to do when they arise, write your rules with these things in mind. For example, if invalid data often appears in a decimal field and should be treated as zero when it does, build a rule that does just that, using the **is_valid** function. Use the **reject** port for unanticipated problems with the data.

- Use prioritized assignment to simplify your rules.

  You might be tempted to write your transformations in the same procedural style you are used to from other systems. This can lead to fairly complicated expressions, such as this:



The above expression can be specified much more cleanly with the following series of prioritized rules:



Activities

Save these activities as **Reformat_#.mp**, where **#** is the activity number. Remember to start by building and running each graph with the sandbox parameter **TEST_FLAG** set to **SmallData**. When you are satisfied that the graph runs correctly, change **TEST_FLAG** to **LargeData** and rerun the graph against the full data volume. Set **TEST_FLAG** back to **SmallData** before proceeding to the next activity.

Recall from Chapter 11 that a very easy way to create a DML file that is closely related to an existing one is to follow these steps:

1. On the **Ports** tab of a component, click **Use another port in graph**.
2. From the **Port in graph** drop-down list, select the port carrying the DML you want.
3. Click **Embed** to make a local copy of the DML file within the component.
4. Click **Edit** to modify the new DML file.

PRACTICAL NOTE

Adding and editing fields

You can add fields to the output record format in the **Transform Editor** simply by dragging a connection across from the **Transform Rules** to the empty space below the last field in the **Output** column. After creating a field in this way, you can change the name by clicking it and editing it. You can also open the **Record Format Editor** by right-clicking the output field and choosing **Edit** from the pop-up menu.

Using the **string_concat** function

The string functions are a set of frequently used built-in functions listed in the **Expression Editor**. The built-in function **string_concat** can be used to combine up to 25 strings.

1. Starting with the **Customers** multifile, create a single field, **fullname**, for each customer. The field should consist of the first name and last name, separated by a space (use **"\001"** as the delimiter for the output field). Do not include the original first and last name fields in the output, but do include the other fields from **customers.dml**.

Calculating derived information

You can use the transform in a REFORMAT component to calculate derived information from multiple fields within the same input record. After building an expression in the **Expression Editor**, click the **Validate** button to check the syntax.

2. Using the **Transactions** multifile, calculate the unit price for each transaction by dividing **transaction_amt** by **quantity**. Put records with invalid transaction amounts in a separate file.

PRACTICAL NOTE

Using "Propagate through" to specify a record format

The **Propagate through** option (on the **Ports** tab of a component) is useful when the output record format of a REFORMAT component is exactly the same as the input record format. Selecting this checkbox causes the **in** port record format to propagate directly to the **out** port. For components like FILTER BY EXPRESSION, the checkbox is selected by default. For components with transforms (such as REFORMAT), select the checkbox if you want to use the same record format for both incoming and outgoing data.

"Cleaning" input data

A common use of the REFORMAT component is to "clean" input data so that all the records conform to the same conventions. For example, the built-in function **decimal_lrepad** can strip leading spaces from a decimal and pad it to a fixed length with a specified character.

3. Make sure the leading zero has not been dropped from any zip codes in the **Customers** dataset. The zip codes should be properly formatted with precisely five characters, all digits.

Changing the case of strings

Another common data quality issue occurs with string data. Such data can arrive using a mixture of conventions, but for comparisons, all the data must be in the same format. You can use the built-in functions **string_upcase** and **string_downcase** to change the case of strings.

4. Make sure the states in the **Customers** dataset have the proper US Postal Service format: two uppercase letters.

Using date and datetime formats

DML offers a number of built-in functions for manipulating **dates** and **datetimes**. The **date_year**, **date_month**, and **date_day** functions extract the indicated parts of a date and return an integer.

5. Using the **Transactions** multifile dataset, put the month and year from the transaction date into separate fields. Discard the day. Put records with invalid dates in a separate file. Do not include the original date field in the output.

PRACTICAL NOTE

Built-in date functions and invalid dates

Using a built-in date function on an invalid date will cause an error, usually resulting in a reject.

SOLUTIONS

The solution graphs are named **solutions/Reformat_#.mp**, where **#** is the activity number.

13

Lookup files

Using a lookup file is a fast and convenient way to make information from a file available to transform functions in any component when the file's data fits in memory. You can use the LOOKUP FILE component (in the **Datasets** folder of the **Component Organizer**) to extend the collection of inputs available for use in a transform function (such as the one that drives a REFORMAT component). Lookup files are often used to store reference data that consists of relatively static business information, such as product descriptions.

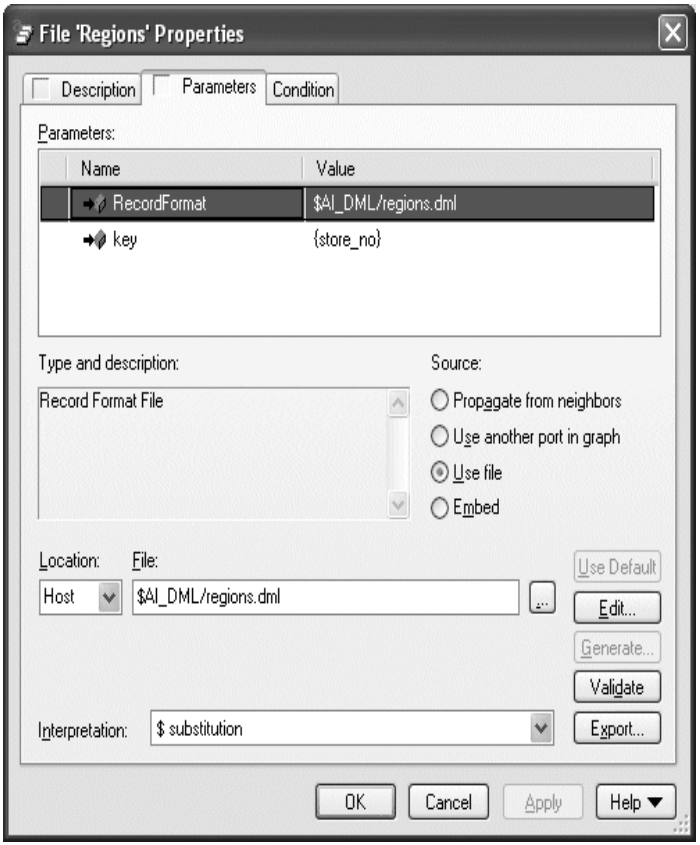
Some developers may assume that only reference tables can be used as lookup files or that the associated data files must be very small. In fact, the only size restriction on a lookup file is that it must fit in the available memory. On a modern server, several gigabytes of memory might be available, so a very large file with millions of records could be used as a lookup file. However, you should be careful when making lookup files out of very large datasets, because other applications can place constraints on memory usage.

You can access a lookup file in any expression. For example, you can use a **lookup** function in the **select_expr** of a FILTER BY EXPRESSION component.

You can also use an INTERMEDIATE FILE or OUTPUT FILE component as a lookup file by selecting the **Add to catalog** checkbox (to the right of the **File type** drop-down list on the **Description** tab of the component). When you do this, the component's **Properties** dialog displays a **Parameters** tab, where you can specify the key. You can then use the file as a lookup file in later phases of the graph.

Configuring lookup files

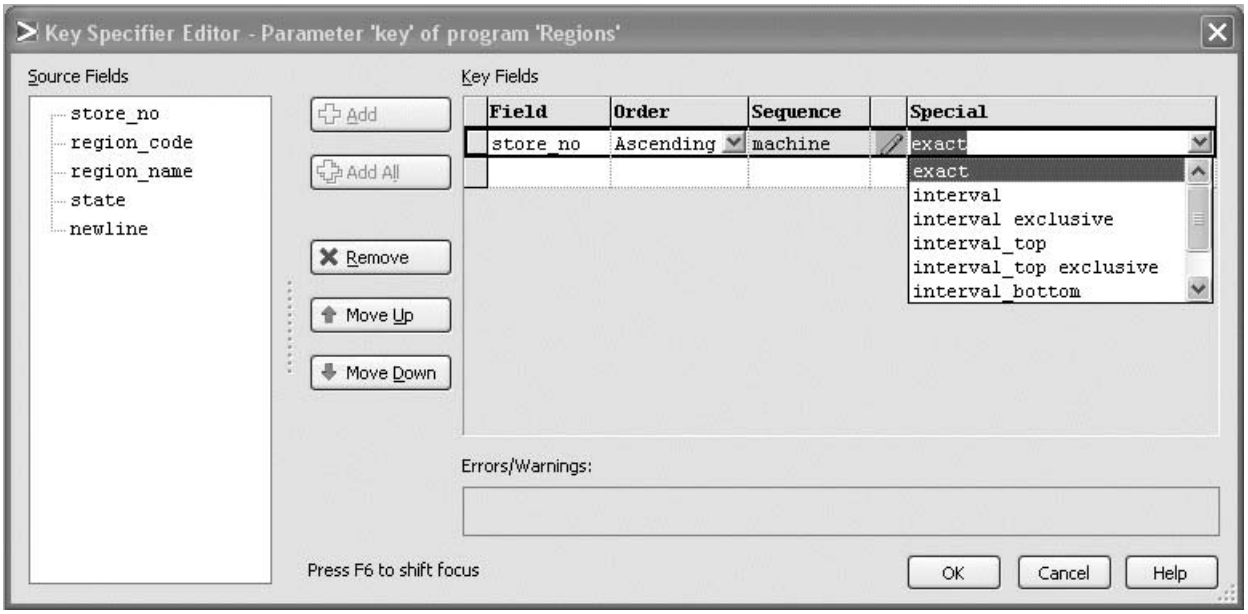
When configuring a lookup file, you specify the record format on the **Parameters** tab, because the LOOKUP FILE component has no ports to hold it. You must also specify the key, which is used to index the lookup file and can consist of one or more fields.



The parameters of a lookup file

- **To select advanced lookup modifiers that define interval ranges:**

  1. On the **Parameters** tab of the LOOKUP FILE component, double-click the **key** parameter to display the **Key Specifier Editor**:



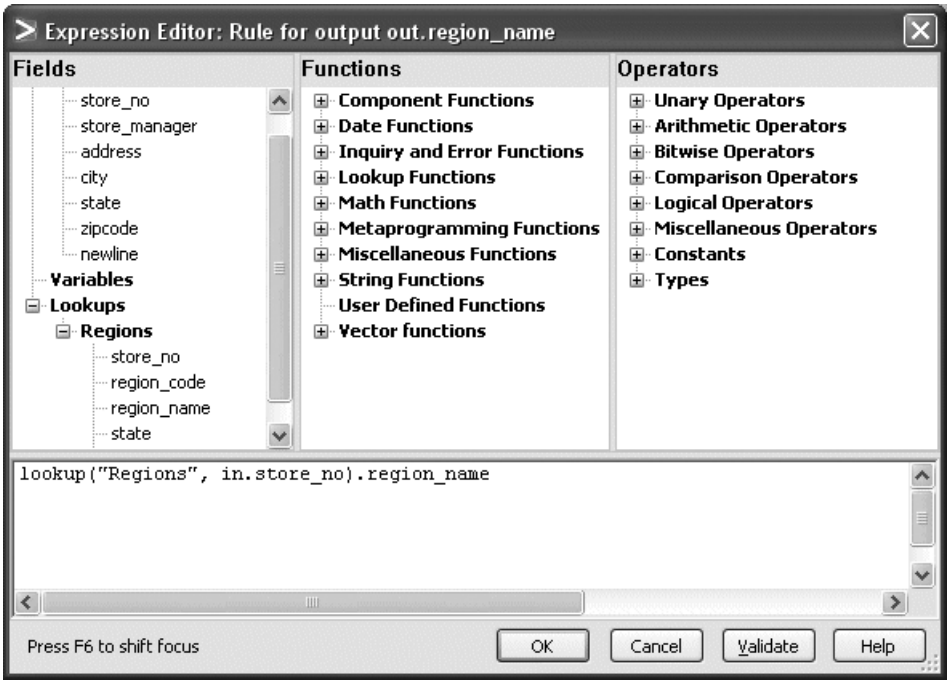
2. In the **Special** column of the **Key Specifier Editor**, select an interval range. (For more information, search for **"interval lookups"** in Ab Initio Help.)

Lookup functions

The **lookup** function and related functions return from a lookup file the whole record whose key matches a given value. Usually, you will want only a single field from the **lookup** call, so you should use the dot (**.**) operator after the call to access the field of interest. For example, if you want the region name for a given store, you can write:

```
lookup("Regions", in.store_no).region_name
```

You can easily create a call to a **lookup** function by using the **Expression Editor**. When you double-click a field in any lookup file in the **Fields** column of the **Expression Editor**, a **lookup** function call with the correct syntax, including *.fieldname*, is automatically inserted. The call to the **lookup** function will have a question mark (**?**) in place of each necessary key argument, and you can then supply the values either by double-clicking the appropriate field(s) or by typing the name(s).



A lookup expression in the Expression Editor

Although **lookup** can be used to do some of the things you can do with a JOIN component, one interesting difference between the two mechanisms is that the key value passed to the **lookup** function does not have to be a field in an existing record, as it does in a JOIN. Instead, you can compute *any* value — based on input fields, results of functions, and so on — to use as the key for a lookup file.

Activities

Save these activities as **Lookup_#.mp**, where **#** is the activity number.

Building an expression that uses a **lookup** function

A convenient way to build an expression that uses a **lookup** function is to use the **Expression Editor** and click the field that you want to retrieve from the lookup file. A question mark (**?**) appears everywhere you need to specify a value for a field in the key. You must supply these arguments in the order in which the fields appear in the key.

1. Using the **Regions** dataset as a LOOKUP FILE, produce a dataset that attaches the store's region name to each of the **Stores** records. (Recall that in Chapter 3, Activity 5, you created a **Regions** dataset using the URL **file:$AI_SERIAL/regions.dat** and the record format **$AI_DML/regions.dml**. The stores listed in the **Stores** dataset are located in regions as specified in the **Regions** dataset.)

Using shared fields as keys

When you access data by using either a LOOKUP FILE or a JOIN, it is essential to know which fields are shared between datasets and can therefore be used as keys. For example, you can connect records in the **Transactions** file to the **Customers** file through the **customer_id** field.

2. Using the **Customers** dataset as a LOOKUP FILE, produce a dataset that attaches to each transaction a field named **customer_name**, consisting of the customer's first name and last name, separated by a space. Use the serial **customers.dat** and **transactions.dat** files. For cases where there is no matching **customer_name** in the LOOKUP FILE, use a default field value.
3. Using the **Products** dataset as a LOOKUP FILE, produce a dataset that includes the product name for each transaction in place of the product code.

Computing the key for a lookup file

The **transaction_amt** and **quantity** from the **Transactions** dataset and **whs_price** (wholesale price) from the **Products** file can be combined to compute a markup percentage:

```
(transaction_amt / (quantity * whs_price) - 1) * 100
```

4. Your company has reason to suspect fraud in cases of extremely high markups. Find all transactions where the markup is more than 300%. (Remember that the **lookup** function can be used in a FILTER BY EXPRESSION component.) Use a TRASH component (and check the tracking data on it) to get a count of transactions whose markup was less than or equal to 300%.

Using more than one key in a single expression with a lookup file

The customer's address is in the **Customers** file, and the city for each store is in the **Stores** file.

5. Find all transactions where a customer bought a product in a city other than the customer's home town. Use the serial **customers.dat** and **transactions.dat** files.

SOLUTIONS

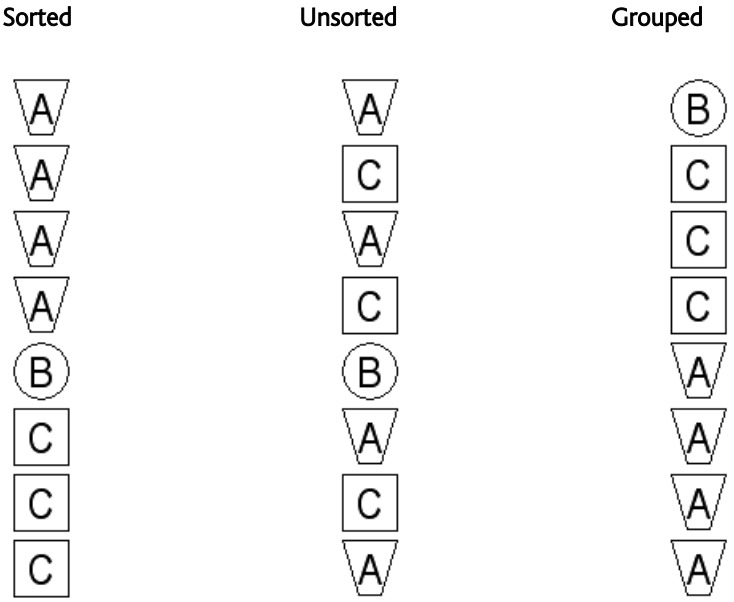
The solution graphs are named **solutions/Lookup_#.mp**, where **#** is the activity number.

14

Aggregating across groups of records: ROLLUP

The ROLLUP component produces a single record for each group of records sharing a common key. This is particularly useful for computing various kinds of summaries — such as totals and averages — and for finding the minimum and maximum for the records in a group. The ROLLUP component can process input data that arrives in any order. The data may be:

- **Sorted** — Presorted into key groups and ordered by key (for example, transactions grouped by store number, with the groups ordered by store number)
- **Unsorted** — With members of the various key groups spread throughout the data
- **Grouped** — Presorted into key groups but not in key order (for example, transactions grouped by store number, but with the groups in no particular order)



BEST PRACTICE

Use components economically.

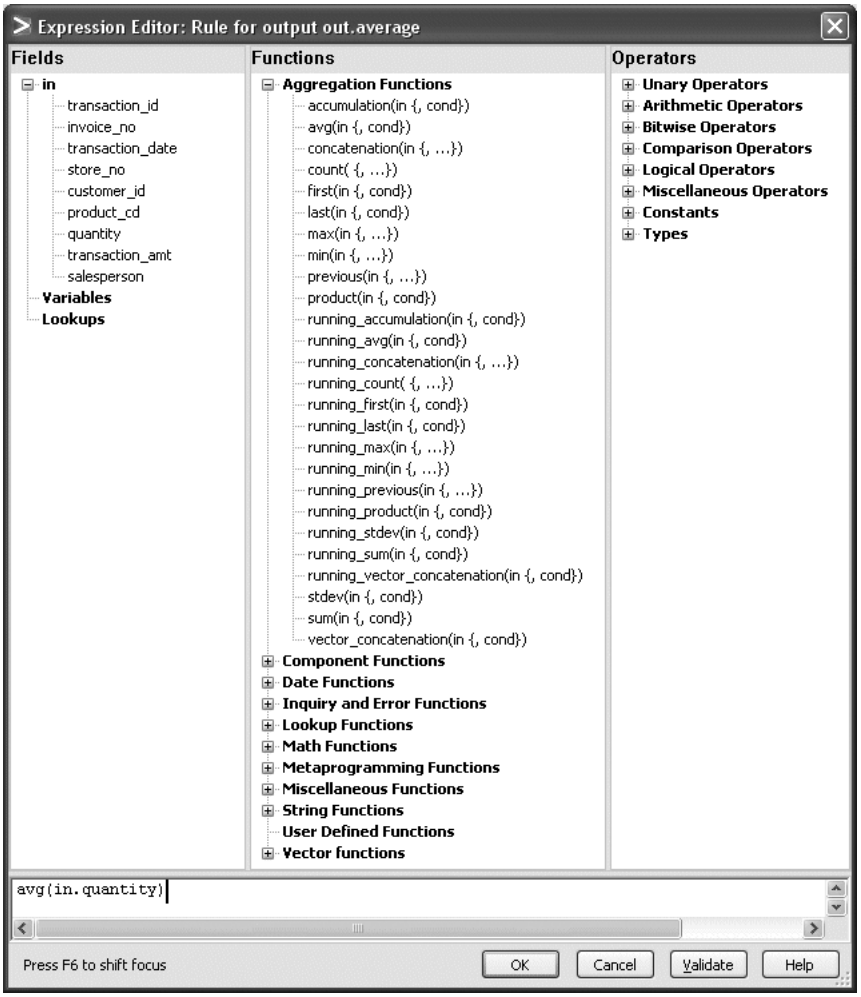
It is seldom necessary to use a REFORMAT component directly after a ROLLUP component. Any transform that a REFORMAT can do (such as adding, dropping, and computing fields) can also be done with a ROLLUP.

The ROLLUP component, like REFORMAT, processes records according to rules specified in a transform function. In its simplest form, the ROLLUP component's transform function uses built-in *aggregation functions* to compute its summaries for the records in a group. The ROLLUP component is in the **Transform** folder of the **Component Organizer**.

▶ **To open the Transform Editor for a ROLLUP component:**

- Hold down the Shift key while double-clicking the component.

When you drill into a rule (right-click and choose **Edit Rule**), the aggregation functions are listed in the **Functions** pane of the **Expression Editor**.



Aggregation functions available in ROLLUP

For information about using the ROLLUP component to do aggregations beyond those available through the built-in aggregation functions, see Chapter 18.

Sorted versus in-memory

The ROLLUP component works with groups of records. You roll up or aggregate on the key that describes those records, which can arrive at the ROLLUP in any of the three different orders described earlier. The order in which the records arrive is a very important distinction for the ROLLUP component, because it has to receive all records for a key group before it can create the single output record for that key group. Thus, knowing your data order is essential to using the ROLLUP component in the most efficient manner.

You must provide information to the ROLLUP component about the order of the input records by using the **sorted-input** parameter (the default is **Input must be sorted or grouped**). For grouped data, you can use the **check-sort** parameter to tell the ROLLUP what to do if it encounters an out-of-order record.

Sorted data

If the input records are sorted by key, ROLLUP keeps track of information for only one key group at a time. As soon as the next group starts arriving, ROLLUP can output its single record for the previous group. Thus, ROLLUP can operate in pipeline fashion, passing results to the next component in a stream while it continues to work on new input. For sorted input data, set **sorted-input** to **Input must be sorted or grouped** (the default).

Unsorted data

If the input records are neither grouped nor sorted, the rollup must be done in memory. Temporary information must be saved for every key group until all the input records have been read, because ROLLUP cannot know that there is not another record for a specific group until all the input is read.

After that happens, the in-memory ROLLUP outputs a record for each group. For input data such as this, set **sorted-input** to **In-memory: Input need not be sorted**. The **check-sort** parameter is not available, since the records are not sorted or grouped.

Grouped data

If the input records are grouped by key but not sorted, set **sorted-input** to **Input must be sorted or grouped** (the default), and set **check-sort** to **False** so that the ROLLUP will operate on one key group at a time but will not abort on the first out-of-order record. This preserves the pipeline, passing results to the next component in a stream while it continues to work on new input.

RESULT ORDERING AND THE **sorted-input** PARAMETER

The setting of the **sorted-input** parameter also affects the result ordering of the ROLLUP component. A ROLLUP with **sorted-input** set to **Input must be sorted or grouped** maintains the order of its input records regardless of the state of the **check-sort** parameter, whereas an in-memory ROLLUP *does not* internally sort the data, nor does it maintain the input order of the key values. Thus, the output of an in-memory ROLLUP is not in sorted order, nor are the key groups guaranteed to be in the same order as they were input.

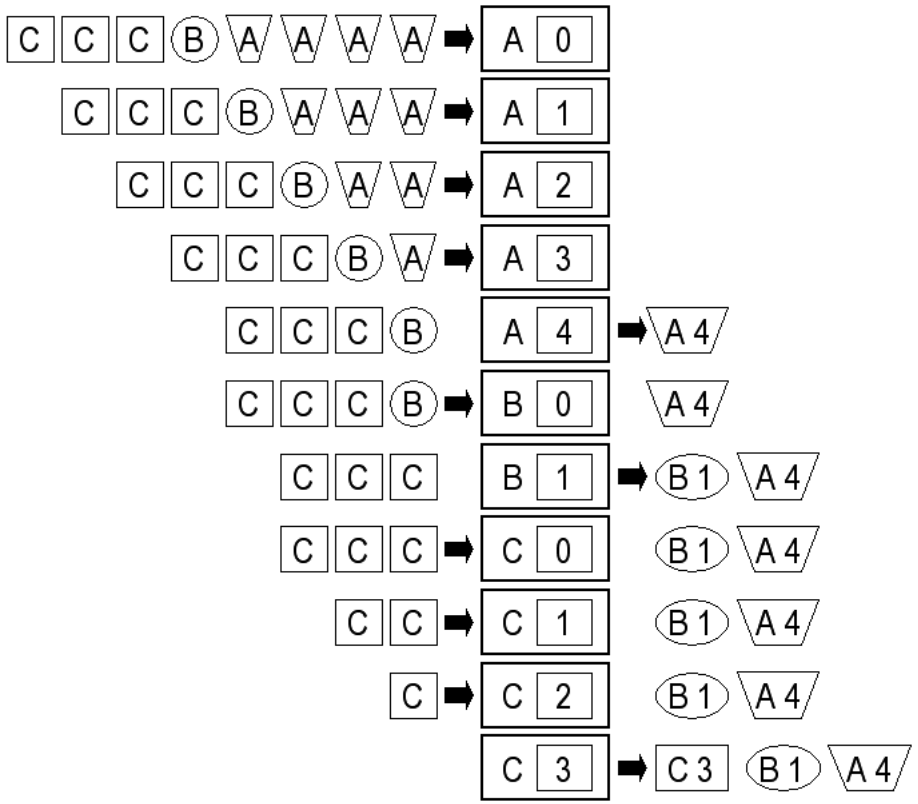
PERFORMANCE IMPLICATIONS OF THE **sorted-input** PARAMETER

The performance implications of the **sorted-input** setting are significant. An in-memory rollup requires more memory but enables you to avoid sorting the input. The amount of memory needed by the in-memory rollup is proportional to the number of key groups in the input; it is roughly the number of key groups times the sum of the size of an input record and the size of the temporary variables used by the aggregation functions. If the amount of memory needed exceeds the **max-core** parameter setting, ROLLUP stores some of its data on disk. If the data volume is large, or if other components in the graph can take advantage of sorted data, you might want to sort the data before rolling up instead of using an in-memory rollup. However, in cases where a ROLLUP will significantly reduce the data volume, the memory required by a SORT might be larger than that required by an in-

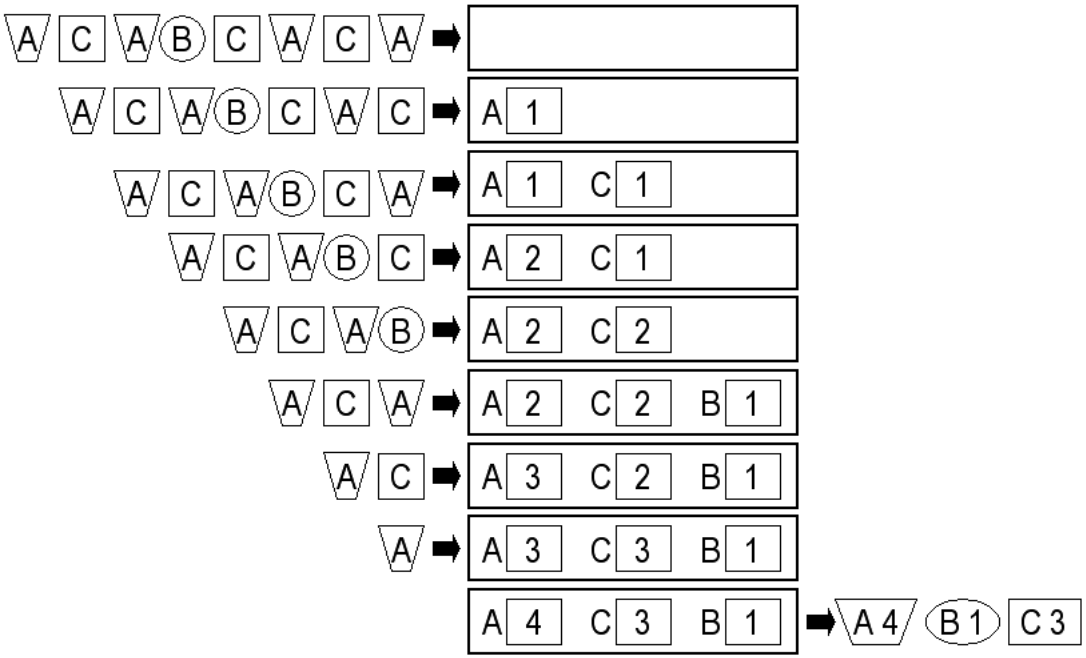
memory ROLLUP. Keep in mind that the ROLLUP will generally produce a significantly smaller dataset than what is presented on the input. Knowledge of what the ROLLUP is producing may drive your implementation.

EXAMPLES: USING ROLLUP TO COUNT RECORDS IN SORTED, UNSORTED, AND GROUPED DATA

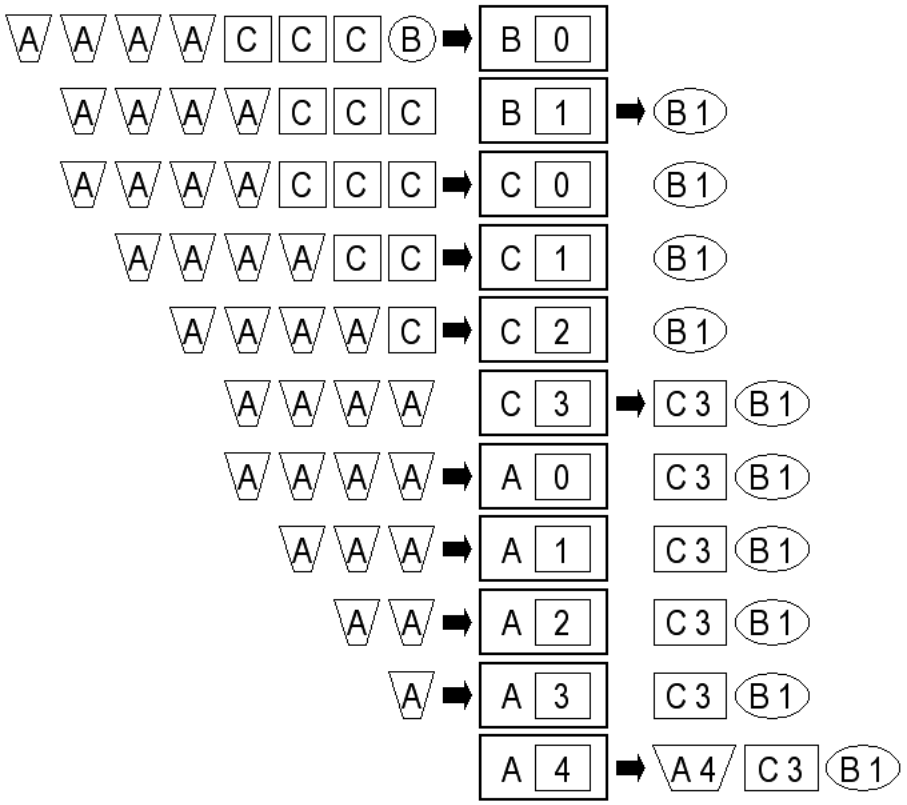
The following three figures show how a ROLLUP uses its **count** aggregation function to compute the number of records with a given key in an input flow of eight records. The arrows on the left show the records being read, while those on the right show the records being output for each key group.



With **sorted-input** set to **Input must be sorted or grouped**, the ROLLUP component maintains a small amount of working memory to compute the result records. As it processes each input record of a group, it updates the working memory. (Here, it is incrementing a counter.) At the end of the group, it produces an output record, clears the working memory, and reuses it for the next group.



With unsorted, ungrouped input, the ROLLUP component maintains a potentially large amount of working memory consisting of a piece of memory for each of a possibly large number of key groups. As it processes each input record of a group, it updates the piece of working memory corresponding to the key of that record. No output is produced and no working memory is freed until all input has been processed.



With grouped input (and with **sorted-input** set to **Input must be sorted or grouped**) and **check-sort** set to **False**, the ROLLUP component maintains a small amount of working memory to compute the result records (as with sorted input). As it processes each input record of a group, it updates the working memory. (Here again, it is incrementing a counter.) At the end of the group, it produces an output record, clears the working memory, and reuses it for the next group.

Activities

Save these activities as **Rollup_#.mp**, where **#** is the activity number.

Key-based activities in parallel are beyond the scope of this chapter; here you will work with the serial datasets **$AI_SERIAL/transactions.dat** and **$AI_SERIAL/customers.dat**. For information about using ROLLUP and JOIN components in parallel, see Chapter 17.

The output record format for a ROLLUP component generally has the key fields the rollup is being computed on (although this is not required), in addition to fields for each aggregation you might compute. Very often you will simply drop input fields that are neither part of the key nor in some way common to the group.

The **count** function

One of the most common uses of the ROLLUP component is to count the number of records in a group. You use the **count** aggregation function to do this. The function counts the number of non-NULL occurrences of its argument. Often a constant is used simply to count the number of records with a given key. For example, **count(1)** counts the number of records for each value of the key specified in the ROLLUP. Similarly, **count(in.part_no)** would count the number of records for which the input field **part_no** is not NULL for each value of the key specified in the ROLLUP.

1. Find the number of transactions made in each store. Use the serial transactions file (**file:$AI_SERIAL/transactions.dat**). When deciding whether to set the **sorted-input** parameter of the ROLLUP to **In-memory: Input need not be sorted** or to add a SORT before the ROLLUP and then set **sorted-input** to **Input must be sorted or grouped**, consider that the number of stores will drive the number of records produced by the ROLLUP. Assume that there are fewer than 300 stores today and that in three years there may be up to 1000. (By Ab Initio standards, even 1000 is a very small number, and a ROLLUP producing so few results will run very well in-memory.)

The **max** function

ROLLUP aggregation functions other than **count** use their argument in a more substantial way, as a value to accumulate or compare against.

2. Use the **max** function to find the largest transaction amount for each **customer_id**. Filter out records with blank transaction amounts beforehand and collect them in a file.

Using multiple aggregation functions

You can use as many aggregation functions as you like in the transform of a single ROLLUP component. Sometimes not every record in a group should be included in a calculation, and if you are using multiple aggregation functions, some might need to ignore values in different records. For example, you might need to take an average excluding invalid values for a given field, but also produce a count that includes records with invalid values. Note that each aggregation function takes an optional second argument that expresses a condition. If the condition is true for a specific value, that value is included in the aggregation. If there are no input records for which the condition is true, the result of the aggregation function will be NULL in most cases (except **count**, where it will be zero). To accommodate this possibility, you can use prioritized rules or default field values.

3. Count the number of transaction records for each customer, the total amount they spent, and the average amount they spent (per record). Use a condition to ignore blank amounts in the average and total.

Dates: **max** and **is_valid** functions

The **max** function, when used on a date field, returns the most recent date. Feeding an invalid date to **max** will generate an error. To avoid such errors, use the **is_valid** function in a condition.

4. Find the most recent transaction date for each customer, considering only valid dates. Also count the number of records with invalid dates for each customer.

Sorted versus unsorted input: **first**/**last** and **min**/**max** functions

Some rollup computations are done differently depending on whether you are processing sorted or unsorted sets of input records in the ROLLUP component. You can use the **first** and **last** functions instead of **min** and **max** if your input records are sorted appropriately. Unlike **min** and **max**, the **first** and **last** functions will not generate errors if they encounter invalid dates, because they simply generate an output record based on the order of records in a group, rather than doing date calculations.

5. Find the earliest and latest transaction date for each customer without using **min** and **max**. Check for invalid dates in the **first** and **last** functions.

Using the empty key to treat all records as a single group

Sometimes it is necessary to treat all records as a single group. You can do this by using the empty key: **{}**. You can create an empty key in the **Key Specifier Editor** by simply clicking **OK** with no key fields selected and then answering **Yes** to the question "Would you like to create a key specifier that treats all records as one group?"

6. Find the largest transaction amount (without sorting the data). Send the records with invalid transaction amounts to a TRASH component so that you can see a count of the bad records without landing them on disk.

"In-memory deduping"

The ROLLUP component can also be used for "in-memory deduping" — that is, for reducing intermixed groups of records that share a common key to a single record per group without first sorting the full dataset (as you must do before using the DEDUP SORTED component). To

dedup in memory, set the **sorted-input** parameter to **In-memory: Input need not be sorted**. Remember that an in-memory ROLLUP does not internally sort the data, nor does it maintain the input order of the key values. You can use the aggregation function **first** or **last**, or simply reference fields of the input record (most commonly with a wildcard rule) to get values directly from one of the records in the group (which record, exactly, is not defined — it might be the first record or the last record in the group, but for capturing values of key fields, any one will do).

7. Create a dataset listing the **customer_id** of every customer who made any transactions. There should be at most one output record for each such customer.

SOLUTIONS

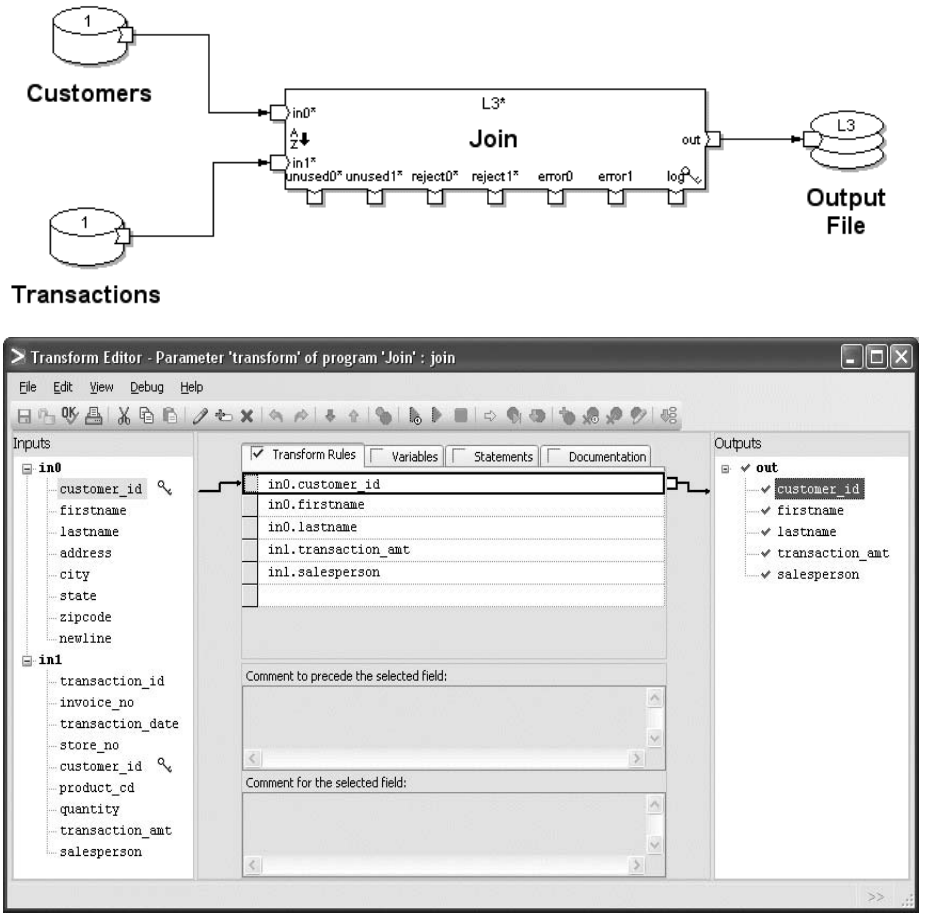
The solution graphs are named **solutions/Rollup_#.mp**, where **#** is the activity number.

15

Combining data from multiple sources: JOIN

The JOIN component (in the **Transform** category of the **Component Organizer**) combines data from two or more flows of records related by key. Nearly every business problem requires the integration of data from multiple sources, and although some important tasks of this kind can be accomplished with lookup files and REFORMAT, in most cases the JOIN component is needed.

The JOIN component matches up records with the same key value from multiple inputs and then uses a required user-defined transform function to combine those records and produce an output record.



A JOIN component and its transform

The JOIN component performs two broad tasks:

- Synthesizes new data from several different sources, usually of differing formats. Output records are built from fields of one input or another, or are computed from fields of several inputs.
- "Differences" or updates records that have the same format. *Differencing* compares two or more flows to identify changes in individual fields or records. *Updating* compares and combines the information in multiple versions of the same set of data to modify an existing set of records or add information to it.

Within these two broad categories, the JOIN component can accomplish a huge variety of work.

PRACTICAL NOTE

Using the JOIN component for database-like operations

The JOIN component can be used to emulate any relational database join involving equal key values. The default for the component is the inner join, but full outer joins, semi-joins, and other database-like operations can also be done efficiently within the JOIN component.

Join types

The many different types of joins depend on the presence or absence of matching records on one or more inputs. You specify the join type by using the **join-type** parameter on the **Parameters** tab for the JOIN component. The following values are possible:

- **Inner Join** — Uses only records with matching keys on all inputs.
- **Full Outer Join** — Uses all records from every input. If a record from one input does not have a matching record in another input, NULL is provided in place of the missing record.
- **Explicit Join** (or *semi-join*) — Uses all the records in one specified input, but records with matching keys in any other input are optional. NULL is used in place of any missing records.

Determining which type of join to use

The most important thing to consider when using the JOIN component is the type of join you need to use. The join types were described above in terms of the input records they use. Now you will look at the join types from the perspective of the output you want to produce.

To understand the three different join types, consider how they work with two input flows (but remember that the JOIN component is not limited to two inputs).

Each of the following examples shows the join type in tabular form. The first two columns represent the records for the data in each of the two input flows (**in0** and **in1**), and the third column represents the result record (**out**). Each row holds the records for any given key value; if a specific key value is not present in one of the inputs or the output, the entry in the corresponding column is **none**.

INNER JOIN

An inner join produces an output record only when all inputs have a record with the specified key value.

IN0	IN1	OUT
key=1 a=3	key=1 b=4	key=1 a=3 b=4
none	key=2 b=5	none
key=3 a=7	none	none

Of the three key combinations shown above, only the one where **key=1** can produce an output record. In the second case, the record on **in1** with **key=2** is considered unused, but you can operate on it by using the **unused1** port of the JOIN component. Similarly, in the third case, the record on **in0** (with **key=3**) is considered unused, but you can operate on it by using the **unused0** port of the JOIN.

FULL OUTER JOIN

A full outer join produces an output record for *any* key value appearing on *any* input.

IN0	IN1	OUT
key=1 a=3	key=1 b=4	key=1 a=3 b=4
none	key=2 b=5	key=2 a=NULL b=5
key=3 a=7	none	key=3 a=7 b=NULL

A full outer join produces an output record for every key combination appearing on the inputs. If there is no matching record on a specific input, JOIN supplies NULL in place of that record in the call to the transform function. Because there are several ways for an output record to be produced by an outer join, the transform rules in the transform function must be written to handle each possible combination of the input records. Generally, prioritized rules are used for this.

NOTE: When you use a full outer join, all records are used; that is, no records are sent to any of the **unused** ports.

EXPLICIT JOIN

With an explicit join, you can use the **record-required** parameters to control whether a record on a given input is required.

With **record-required0** set to **True** and **record-required1** set to **False**, a record is required on input **in0**, but the presence of a record on input **in1** with the same key is optional. This is a left outer join (or, more generally, an **in0**-outer join):

IN0	IN1	OUT
key=1 a=3	key=1 b=4	key=1 a=3 b=4
none	key=2 b=5	none
key=3 a=7	none	key=3 a=7 b=NULL

In the second case above, the record on **in1** with **key=2** is considered unused, and you can operate on it by using the **unused1** port of the JOIN.

With **record-required0** set to **False** and **record-required1** set to **True**, a record is required on input **in1**, but the presence of a record on input **in0** with the same key is optional. This is a right outer join (or, more generally, an **in1**-outer join):

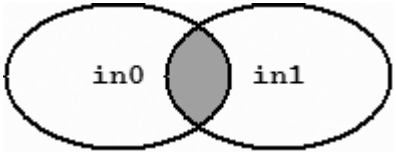
IN0	IN1	OUT
key=1 a=3	key=1 b=4	key=1 a=3 b=4
none	key=2 b=5	key=2 a=NULL b=5
key=3 a=7	none	none

In the third case above, the record on **in0** with **key=3** is considered unused, and you can operate on it by using the **unused0** port of the JOIN.

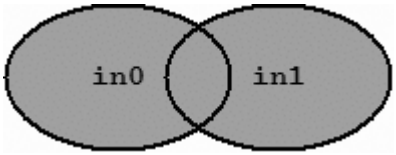
A graphical representation of join types

The diagrams in this section represent the intersection of two datasets based on the values of their keys. Each oval represents all the key values in a given flow. Where two ovals overlap, there are key values that are present in both flows. Output records are produced only for the key values in the shaded regions of each diagram.

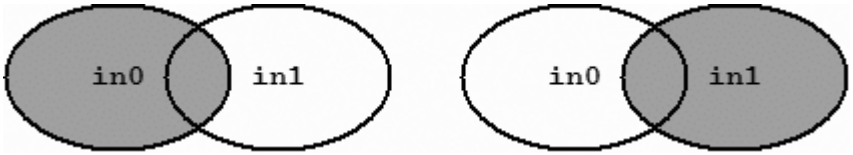
For an inner join, output records are produced only for key values that are common to *both* inputs. The set of keys in the output is the intersection of the keys in both inputs:



For a full outer join, output records are produced for all key values that are present in *either* input. The set of keys in the output of the full outer join is the union of all input key values:



For an explicit join, output records are produced for all the key values in one of the input flows. Thus, with two input flows, there are two types of explicit join. In the following figure, the diagram on the left is for an explicit join where **record-required0** is set to **True**, and therefore output is produced for every key value found in input **in0**. Similarly, for the diagram on the right, **record-required1** is set to **True**, and the output contains all key values found in input **in1**.



Duplicate records and Cartesian products

When using the JOIN component, you should always be wary of the possible presence of duplicate key values in the inputs. If a key value appears multiple times in an input flow, there might be multiple output records for that key value, depending on the join type. This can be a problem if the business requirement is to produce records with unique key values.

The following table shows the output for an inner join where there are duplicates in both inputs:

IN0	IN1	OUT
key=1 a=3 key=1 a=4	k=1 b=4	key=1 a=3 b=4 key=1 a=4 b=4
key=2 a=2	key=2 b=5 key=2 b=6	key=2 a=2 b=5 key=2 a=2 b=6
key=3 a=7 key=3 a=8	key=3 b=9 key=3 b=1	key=3 a=7 b=9 key=3 a=7 b=1 key=3 a=8 b=9 key=3 a=8 b=1
none	key=4 b=9	none
key=5 a=8	none	none

If there are duplicates of a key value in only one input, the number of output records for that key value remains proportional to the number of input records on the port that has duplicates. Each record with the key value in one input matches with the single record that has the same key value in the other input. This is the case with key values **1** and **2** in the table above.

However, when there are records with the same duplicate key value in more than one input, the result is a Cartesian product: the number of output records for the key value is the product of the number of identical key values in each input. This is the case with key value **3** in the table.

You should always compare the number of records on the output of a join with the number of records on the inputs. If you find that there are more records on the output than you expected, check the inputs for duplicates. Even when the input data is "guaranteed" to have no duplicates, it often does. While the JOIN component has a built-in deduplication capability (the **dedup0** and **dedup1** parameters), you should avoid using this capability unless you fully understand the duplicates in the flows you are joining.

Using prioritized rules

When using a full outer or explicit join, you may need to create prioritized rules for the following reasons:

- More than one combination of input records can produce an output record.
- An input record for each key value produced in the output may not always be present on all inputs.

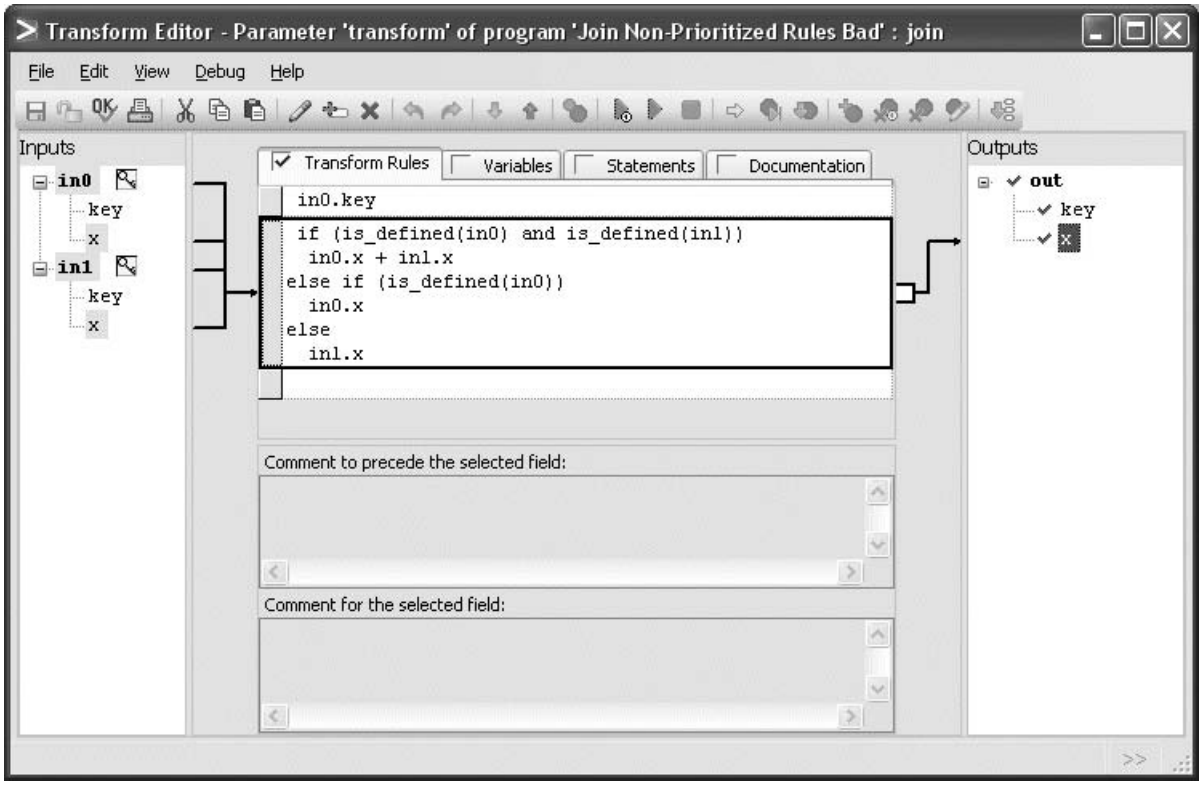
Prioritized rules enable you to choose the order in which transform rules that assign to the same output field are evaluated. For example, if a rule for an output field in the transform function results in NULL because a record on that input is not present for a specific key value, the transform moves on to the next rule for that output field.

In DML expressions, NULL is "contagious" — most kinds of expressions simply return NULL if any part of them evaluates to NULL. For example, the expression **in0.x** returns NULL if **in0** is NULL or if the field **x** of **in0** is NULL; when the field extraction operator encounters NULL on the left of the dot operator (.), it returns NULL because no part of NULL can be anything other than NULL. Similarly, the comparison expression **in0.x > in1.x** returns NULL if either **in0.x** or **in1.x** is NULL. This feature works extremely well with prioritized assignment, so you can write direct and simple transform rules.

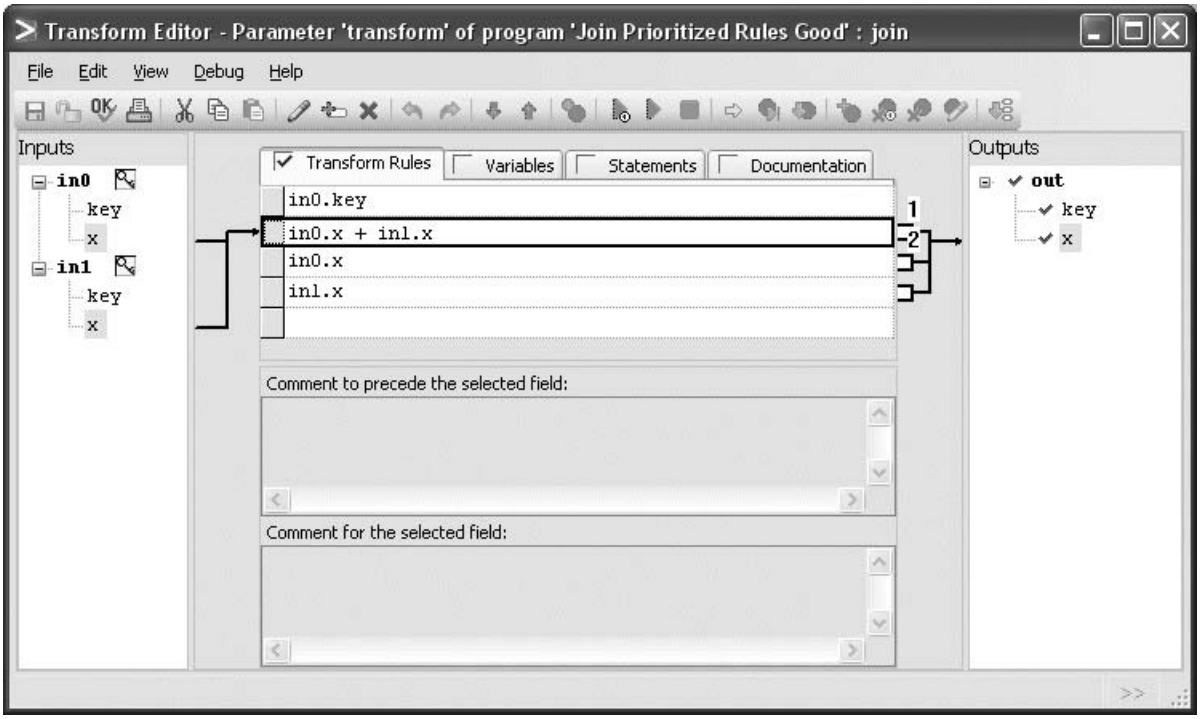
Consider a simple example for a full outer join. Suppose that in addition to the key field there is another field **x** on both inputs, and you want the result field **x** to hold the sum of the input **x** values if both inputs are present, or the sole value if only one is present:

IN0	IN1	OUT
key=1 x=5	key=1 x=4	key=1 x=9
none	key=2 x=5	key=2 x=5
key=3 x=7	none	key=3 x=7

This logic *could* be encoded as a single, complex rule like this:



However, the logic is more clearly expressed as a series of prioritized rules:



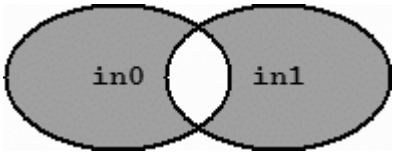
Additional ports

The optional ports on the JOIN component include multiple **reject** and **error** ports — one for each input port. If an error is detected during execution of a JOIN component's transform function, the current record on each of the input ports is rejected. As with other Ab Initio transformation components, if the **reject-threshold** parameter is set to **Abort on first reject**, the process is immediately stopped and an error is reported. Conversely, if **reject-threshold** is set to **Never abort**, the rejected records are sent to their respective **reject** ports: **in0** to **reject0** and **in1** to **reject1**.

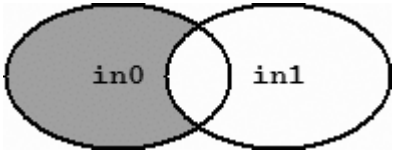
When one or more input flows have records with duplicate key values, the transform function is called once for each combination of inputs. If a specific call results in an error, *all* input records involved in that call will be rejected. But if there are other input records with the same key value, one or more of those "rejected" records will be used again in subsequent calls to the transform function. In the worst case, there can be as many rejected records on a given input as there are calls to the transform function, meaning that the same input record can occur more than once in the reject flow and you may end up with much more rejected data than input data.

The JOIN component also includes one **unused** port for each input port. The records that flow out of the **unused** ports are those with key values that did not match the key values of records on the other inputs.

The **reject** port can be used as an additional flow path, and you can use the **reject** and **unused** ports individually and collectively to do complex join-related computations. For example, you can find all the records with key values that are present in **in0** or **in1** but not in both **in0** and **in1** by gathering all the unused records from an inner join. The following diagram shows this "anti-inner join" (which is the complement image of the diagram for the inner join):



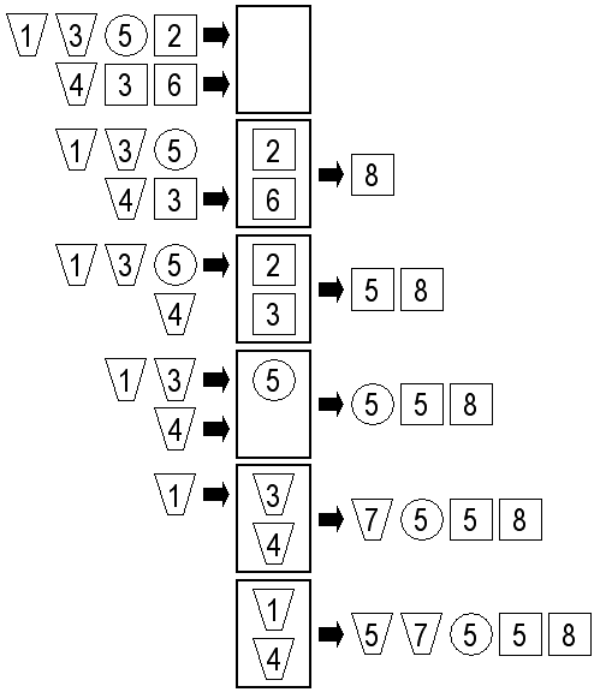
As another example, consider having to find all the records with key values that are in **in0** but not in **in1**. You can do this by using an explicit join where records on **in0** are required, together with a transform that has a rule that calls the **force_error** function if there is a record on both inputs. You must also set the **reject-threshold** parameter to **Never abort**. The records you want will appear in the output flow. The diagram for this join is:



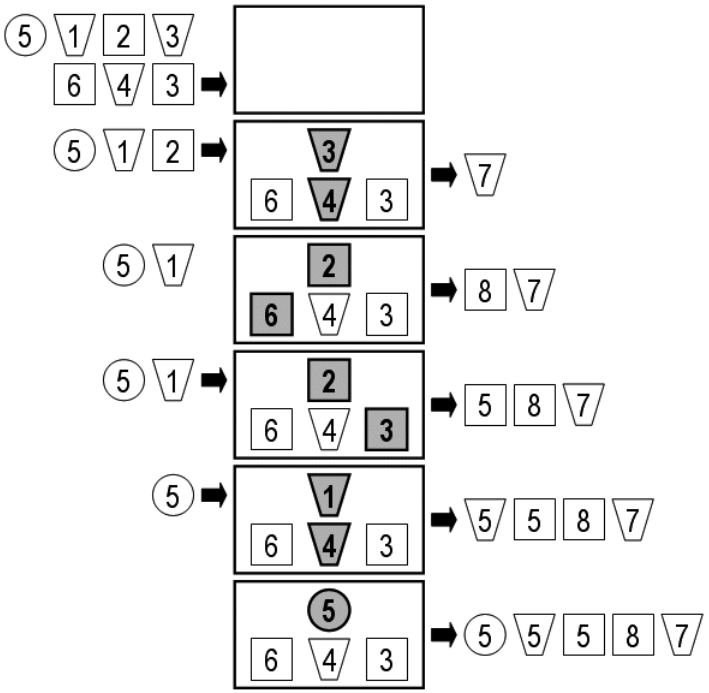
Sorted versus in-memory

By default, the JOIN component requires input data that is sorted on the specified key, and it uses a modest amount of memory. If the input data is not sorted, you can set the **sorted-input** parameter to **In-memory**. In this case, one input is treated as the driving input (usually the input with the largest volume in terms of bytes): to specify the driving input, use the **driving** parameter. This parameter is present only for an in-memory join, and the default setting is the first (top) input, **in0**. All records in the nondriving inputs are loaded into memory before the driving input is read. If all the nondriving inputs plus overhead cannot fit in the amount of memory specified by the **max-core** parameter, input data is staged to temporary files on disk.

Both of the following figures show a full outer JOIN summing the values from two input flows consisting of four and three records, respectively. Shapes represent the key of each record, and the arrows on the left show the records being read into memory, while those on the right show the records being output after the JOIN sums the values of **in0** and **in1** for each key group.



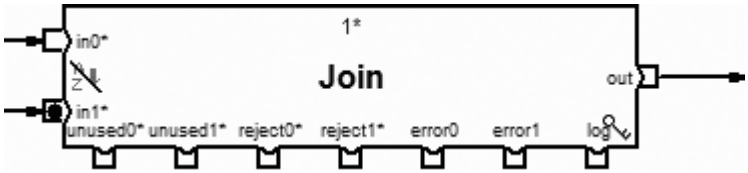
With *sorted* input, the JOIN component maintains a small amount of working memory to compute the result record for each combination of input records. Shapes represent the key of each record.



With *unsorted* input, the JOIN component uses memory proportional to the size of the nondriving inputs. Here the driving input is the top flow, so the bottom flow is read into memory at the beginning of execution. Shapes represent the key of each record.

By definition, the output flow will not necessarily be in the order of the top flow. If you want the output order to be maintained based on the driving flow, you can set the parameter **maintain-order** to **True** (the default is **False** and is the more efficient of the two choices).

If the data volume on nondriving inputs is large, or if other components in the graph can take advantage of having the data sorted, you might want to sort the data before joining instead of using an in-memory join.



A dot on the input port indicates the driving input of an in-memory join.

Override keys

When key fields have different names on different inputs, you can specify the alternate names in the **override-key** parameters. For example, if you are joining two data streams by an account number, and **in0** calls the account number field **acct_id** but **in1** calls it **acct_no**, you can set the **key** parameter to **acct_id** and set **override-key1** to **acct_no**.

Activities

Save these activities as **Join_#.mp**, where **#** is the activity number. Before running the graphs, change **TEST_FLAG** to **LargeData**.

Combining two or more data flows using the JOIN component is a key-based activity. Doing key-based activities in parallel is beyond the scope of this chapter; here you will work with the serial datasets in **$AI_SERIAL**. For information about using ROLLUP and JOIN components in parallel, see Chapter 17.

Attaching fields from one dataset to another

The most straightforward use of the JOIN component is to attach fields from one dataset to another. Suppose you want to attach the appropriate **region_code** (from the **Regions** dataset) to each record in the **Stores** dataset. The **Stores** file contains a single record for each store (identified by **store_no**). The **Regions** dataset contains region information (name and code) for each store, and thus it also contains the **store_no** field. There are no duplicates (with respect to the key **store_no**) in either file, and an output record is needed for every store. Therefore, you can use an inner join to add the required information from the **Regions** dataset to the records in the **Stores** dataset.

Also, since both datasets are presorted on **store_no**, you can use the JOIN component with **sorted-input** set to **Inputs must be sorted**. It is not necessary to sort the data again or set the JOIN's **sorted-input** parameter to **In-memory: Inputs need not be sorted**.

1. Do an inner join with sorted inputs to attach the **region_code** from the **Regions** dataset to each **Stores** record.

Simple differencing

A simple example of a differencing activity is to look for the keys that two datasets with the same record format have in common. The **Customers** dataset has a single record for each customer. Suppose there is another dataset, **Customers Delta**, containing inserts (new customers) and updates (new information for existing customers).

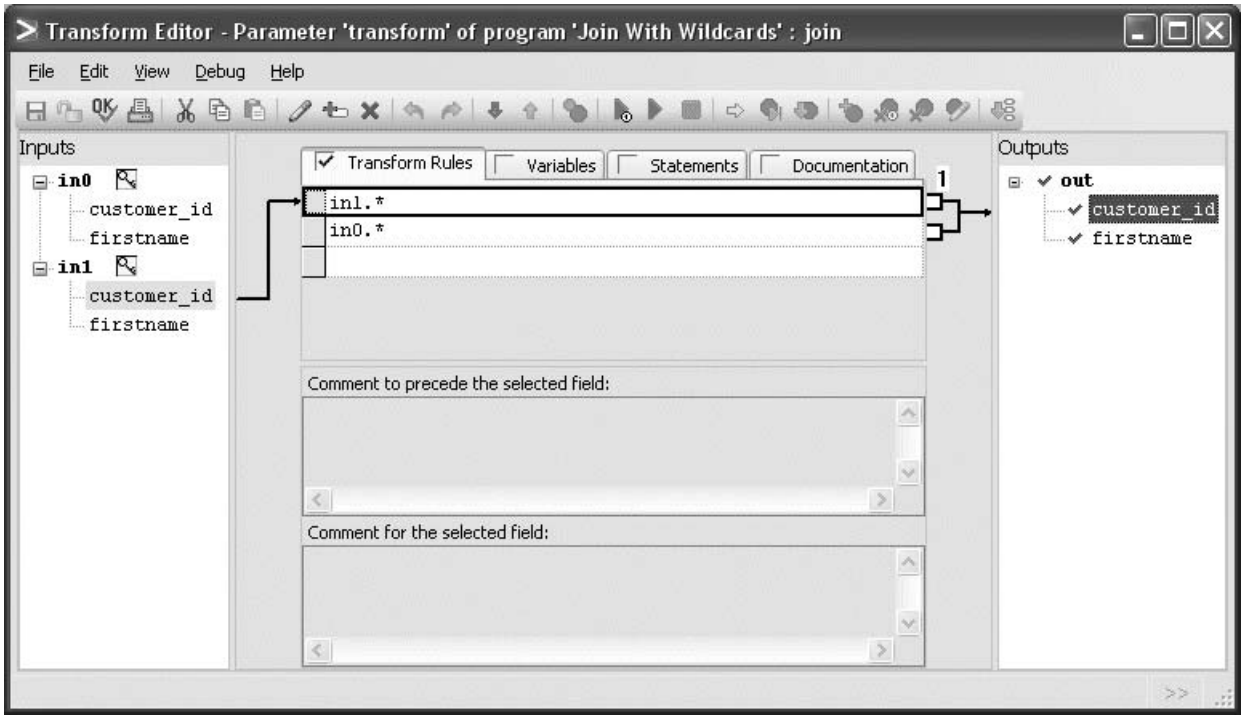
2. Use an in-memory inner join to find the customers present in both the **Customers** file and the **Customers Delta** file. Both files (**$AI_SERIAL/customers.dat** and **$AI_SERIAL/customers-delta.dat**) use the same record format, **$AI_DML/customers.dml**. Use TRASH components to determine how many unused records came from each input. The unused records coming from the **Customers Delta** file will be all the new customers.

More complex differencing: Using a wildcard rule

Now consider a more advanced updating activity using two inputs with the same record format. Suppose you want to create a new **Customers** dataset containing the union of the customers in the **Customers** file and those in the **Customers Delta** file. If a customer appears in both files, you want to take the record from the **Customers Delta** file. You can determine the type of join that is needed by constructing a table that covers all the cases of interest. If the **Customers** file is **in0** and the **Customers Delta** file **in1**, the table looks like this:

IN0	IN1	OUT
key=1 x=5	key=1 x=4	key=1 x=4
none	key=2 x=5	key=2 x=5
key=3 x=7	none	key=3 x=7

Because an output record is produced for each possible input combination, this must be a full outer join. Additionally, when both records are present, the data from **in1** (inserts and updates) is preferred, so the transform must use prioritized rules. Rather than making many pairs of rules, one for each field of the record format, you can use a wildcard rule. To build a wildcard rule in the **Transform Editor**, simply enter an expression like **in0.*** in the rule cell, but do not connect it to the output. A wildcard rule matches any fields not explicitly assigned. In general, different wildcard expressions can match different sets of rules, and the GDE does not do its automatic assignment of priority to such rules. Thus, you must set the priority explicitly by using the pop-up menu on the rule. In the **Transform Editor**, these rules look like this:



3. Use the following business rules to produce an updated **Customers** dataset using the **Customers Delta** file **$AI_SERIAL/customers-delta.dat**:
   - If a new/updated record matches an existing record, take the new/updated record.
   - If a new/updated record does not match an existing record, keep the new/updated record.
   - If an existing record does not match a new/updated record, keep the existing record.

Prioritized rules

Now refine the previous graph to update only the address information for each customer (address, city, state, and zip code). The join type will remain the same, but the transform must be changed. Before assigning an address field from a record in the **Customers Delta** file, you should check to see whether the field is blank. If it is, the field from the existing record should be used whenever such a record exists. You will need individual prioritized rules for the fields of interest, but you can still use a wildcard rule to take care of the others.

4. Use the following modified business rules to update the **Customers** dataset using the inserts and updates file **$AI_SERIAL/customers-delta.dat**. Consider these three cases:
   - A new/updated record matches an existing record. For each field in the address, if the new/updated value is not blank, use it in the output record; otherwise, use the value from the existing record.
   - A new/updated record does not match an existing record. Keep the new/updated record.
   - An existing record does not match a new/updated record. Keep the existing record.

   Now fill in the address information that is missing in the **Customers Delta** file by using the existing information from the **Customers** file. This time you will be producing output records for customers present in the **Customers Delta** file.

   Suppose you want to produce an output record for all key values that are in the **Customers Delta** file, but not for key values that are present only in the **Customers** file. This is an explicit join with records required on the input port that **Customers Delta** is connected to. Note that if there is no existing record, you will use a field from the **Customers Delta** file, even if it is blank.

5. Use an explicit join to produce a dataset that fills in missing address information in the **Customers Delta** file (**$AI_SERIAL/customers-delta.dat**) with information from the **Customers** file (**$AI_SERIAL/customers.dat**). Consider these three cases:

- A new/updated record matches an existing record. For each field in the address, if the new/updated value is not blank, use it in the output record; otherwise, use the value from the existing record.
- A new/updated record does not match an existing record. Keep the new/updated record.
- An existing record does not match a new/updated record. Save to TRASH so that the record count is recorded.

Using the **unused** and **reject** ports

It is not always possible to get the records you want as the output of a basic join type. However, you can greatly extend the functionality of the JOIN component by using the **unused** and **reject** ports. There is also often more than one way to accomplish a join that falls outside the basic join types. You can find records with key values that are in one input or another but not in all by collecting the unused records from an inner join. The same results can be found as the output of a full outer join if records for key values that are present on all inputs are rejected using **force_error**. Remember that even if you are not using a basic join type, you must still pay attention to duplicates.

6. Join the **Customers** file to the **Customers Delta** file to identify the unchanged customers in the existing file and, separately, the new customers in the **Customers Delta** file.

PRACTICAL NOTE

Discarding data from a port that requires a flow connection

It is not necessary to write the output of a JOIN to a file. However, the **out** port is a required port, which means that some component must be connected to it. In cases where you don't need the output data, you can connect a TRASH component to the required port.

Working with duplicates

Another use of JOIN is to create a list of all occurrences of a certain value or event; however, this can be complicated by the presence of duplicates. Recall that for the activities in this chapter, you are using the serial datasets. Up to this point, you have not done a join using the large **Transactions** file (**$AI_SERIAL/transactions.dat**). This dataset is indexed by the **transaction_id**, but it also contains the **customer_id**, **product_cd**, and **store_no** fields that connect it to all the other datasets. However, the **transaction_id** is the only field that has a unique value for each transaction record. If you use any of the other fields listed above as a

key, key values will match multiple records, and the **Transactions** dataset will effectively have duplicate keys. Consider the task of identifying all products that were sold in February 1999. Each product should appear only once in such a list; therefore, transaction records with duplicate **product_cd** values should be removed before joining.

In the following activity, you should remove the duplicates (based on the **product_cd** key) from the transaction data before the JOIN component, and filter the data to find the February 1999 transactions before the JOIN.

7. Build a dataset containing all products that were purchased in quantities greater than **1** (in any given transaction) in February 1999. The graph should also produce a dataset of products that do not meet that criterion. Develop and test with **TEST_FLAG** set to **SmallData**, and then test with **TEST_FLAG** set to **LargeData**.

   Implementing some business requirements removes duplicates as a side effect. For example, the ROLLUP component produces one record per group that is identified by a unique key value. Consider the task of attaching transaction summary information to each record in the **Customers** file. If you roll up the **Transactions** file on the **customer_id** to produce this summary information, there will be only one summary record per customer going into the JOIN. However, if the customer did not make any transactions, there will not be a summary record. If you want to attach this summary information to every record in the **Customers** file, which type of join should you use? The table below summarizes the join logic, with the **Customers** file as **in0** and the transaction summary records as **in1**:

IN0	IN1	OUT
key=1	key=1 summary=123	key=1 summary=123

IN0	IN1	OUT
none	key=2 summary=456	none
key=3	none	key=3 summary=0

This is an explicit join with records required on **in0**. You should use prioritized rules to assign a value when the transaction summary is missing.

8. Produce a new dataset that attaches the total amount of purchases each customer made in 1999 to records from the **Customers** file. If a customer did not make any purchases in 1999, use **0.00** as the total amount.

SOLUTIONS

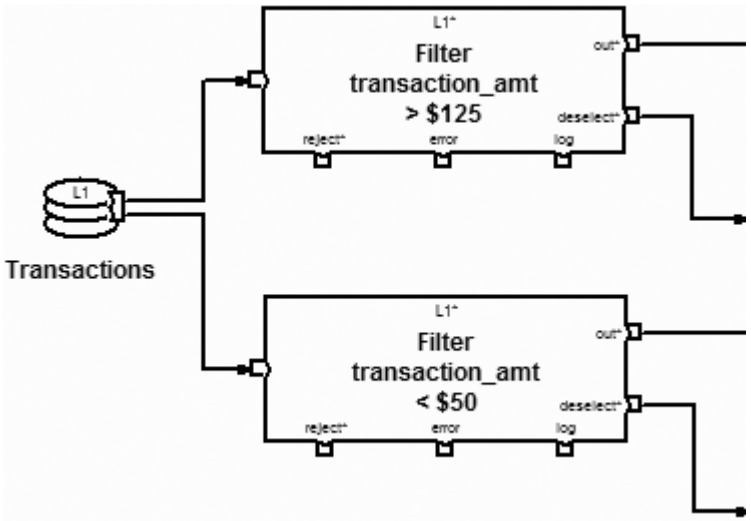
The solution graphs are named **solutions/Join_#.mp**, where **#** is the activity number.

16

Parallelism

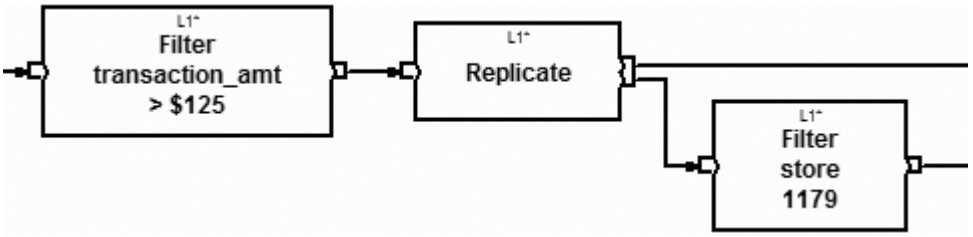
Applications built with Ab Initio software can exhibit three forms of parallelism: component parallelism, pipeline parallelism, and data parallelism.

Component parallelism occurs when more than one component is running at the same time on different streams of data. For example, you could use two FILTER BY EXPRESSION components at the same time to find transactions greater than and less than certain amounts:



Component parallelism: both FILTER BY EXPRESSION components can operate simultaneously.

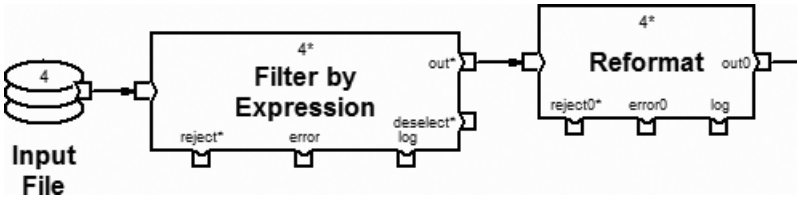
Pipeline parallelism occurs when two or more components process different parts of the same data stream at the same time. This can happen when the first component produces output records before it has finished reading all its input. This enables the next component to begin processing before the first one has finished.



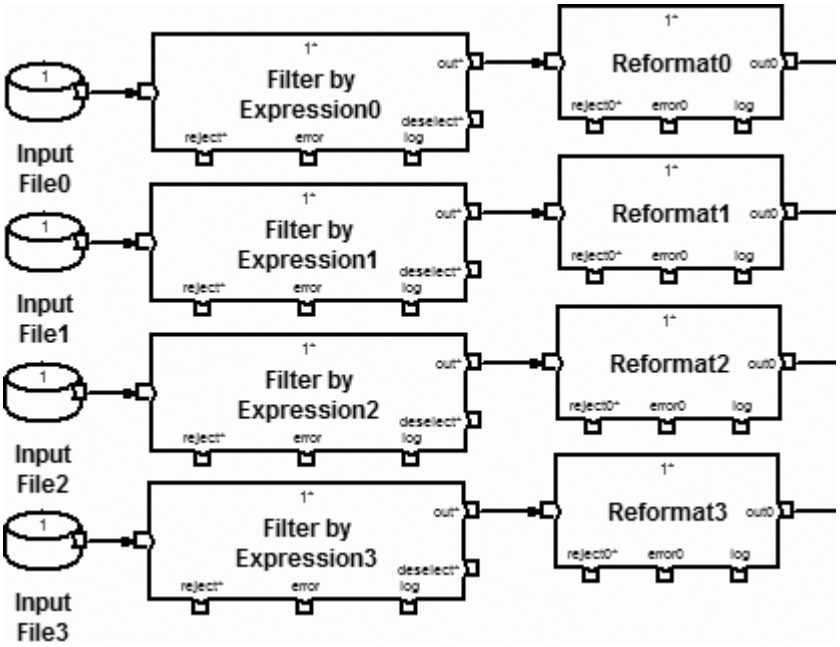
Pipeline parallelism: the three components process different parts of the same data stream.

Any component that must read all its input before producing any output is said to "break" pipeline parallelism, because the next component must wait for that component to finish before starting. For example, a SORT component always breaks pipeline parallelism: it must see all the records before producing any output, because the last record it reads might be the first one in the sort order. Similarly, a ROLLUP component whose **sorted-input** parameter is set to **In-memory: Input need not be sorted** breaks pipeline parallelism, as its results cannot be produced until the input stream has been completely consumed.

Data parallelism occurs when multiple copies of a process act on different subsets of data at the same time. The processing time of a single dataset is typically proportional to data volume, so by dividing a dataset into smaller pieces that can be processed independently you can process the whole dataset faster using multiple CPUs at the same time. Ab Initio software provides a convenient way to run multiple copies of a process using parallel datasets called *multifiles* (see the next figure). In many of the activities you have already done, the data is stored in a two-way multifile. For example, when you were filtering in Chapter 5, your graphs were running in parallel.



Data parallelism: a four-way multifile is read and processed by a FILTER BY EXPRESSION and a REFORMAT, each running four-ways parallel. Here, data parallelism and pipeline parallelism combine: two components, each with four parallel instances, give a total parallelism of eight.



An expanded view of the multifile and data-parallel components shown in the preceding figure. Each row represents one partition of processing.

Multifiles

A *multifile* is made up of a set of serial files (*data partitions*) and a control file that lists their locations. You refer to a multifile set by giving the URL of the control file, often referred to informally as the multifile. Multifiles reside in multidirectories within a multifile system; these are analogous to serial directories and serial filesystems, and in fact are built on top of those. Often, project-level multifile systems are set up by an Ab Initio administrator when development begins.

Layout

The *layout* of a component determines where a component process physically executes and where any temporary files used by the component are stored. By default, layout is propagated from neighboring components. For example, a FILTER BY EXPRESSION connected to an INPUT FILE takes its layout from that INPUT FILE.

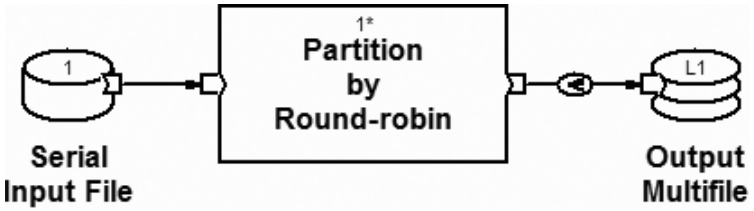
In some situations you might want or need to set the layout explicitly. If there is no obvious layout to choose (as is the case when a component is positioned between two partitioners, or between a partitioner and a departitioner), the layout cannot be inferred from neighbors and must be specified explicitly. To set a component's layout, open the component's **Properties** dialog, select the **Layout** tab, and make a choice. The most common choices after **Propagate from Neighbors** are **URL** (which specifies an explicit location by URL) and **Component** (which ties the component's layout to another component). In a URL layout, the URL can be a serial URL (such as **file:$AI_SERIAL**) or a multifile URL (such as **mfile:$AI_MFS**).

Partitioning

When you use multifiles, some operations — simple, record-at-a-time operations like filtering and reformatting — provide the same results regardless of partitioning. However, other operations, like ROLLUP and JOIN, are sensitive to partitioning because they require records with the same key to be brought together in the same place. In this chapter you will learn how to partition data, including how to partition it so that like-keyed records are brought into the same partition.

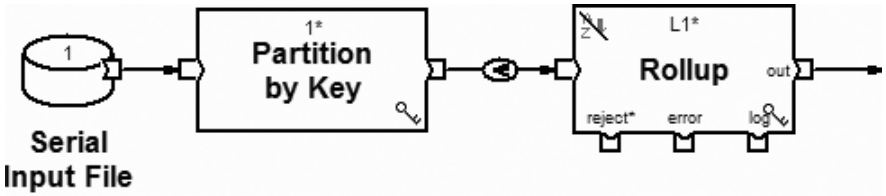
A serial file effectively consists of a single partition. To construct a multifile or a partitioned flow from a serial file, you use a component from the **Partition** category of the **Component Organizer**. The two most commonly used partitioning components are:

- PARTITION BY ROUND-ROBIN — Distributes the data evenly among partitions in a fashion analogous to dealing cards: each record in succession is passed to each partition in succession. The resulting distribution is as even as possible.



- PARTITION BY KEY — Directs records to partitions according to the key of each record. Each key value is always sent to the same partition, so records with the same key value are guaranteed to go to the same partition. This is critical to getting the correct results in most situations when you are using components such as ROLLUP and JOIN, which rely on records that have the same key being together in the same partition. An internal algorithm determines which partition a record is sent to, based on the value and data

type of the key. If the number of key groups is much larger than the number of partitions and no key group has significantly more records than the others, the records will be fairly evenly distributed among the partitions.



For the activities in this chapter and Chapter 17, the multifile **Transactions** dataset is partitioned by key on **transaction_id**, and the multifile **Customers** dataset is partitioned by key on **customer_id**.

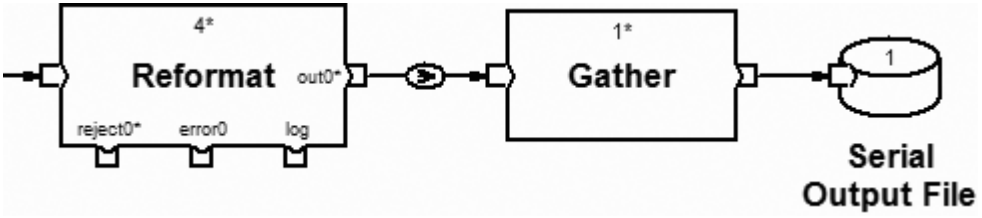
The presence of abnormally large key groups can give rise to *data skew* — some partitions having more data than others. Data skew can reduce actual performance from its theoretical peak because some CPUs might be given more work to do (more data to process) than others. In extreme cases, a process that was intended to be parallel might effectively run serially because all the data is in one partition and therefore all the work is done by one CPU. Skew can often be reduced by a better choice of key or by separating out the large key groups for separate processing.

Departitioning

The components in the **Departition** category of the **Component Organizer** read data from multiple partitions to produce a single partition. The two most frequently used departitioners are:

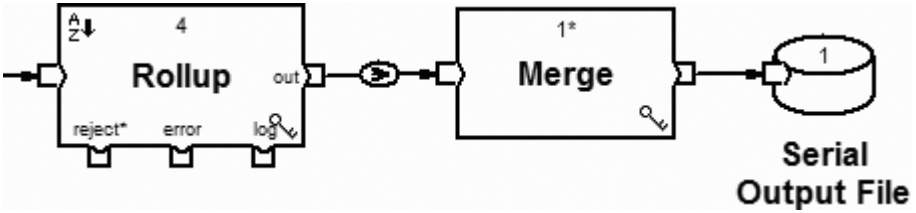
- GATHER — Reads data records from multiple partitions as they become available. It does not impose any kind of ordering on records from different input partitions, and will not necessarily produce the same order of records every time. Many components have this

fan-in capability built in, so an explicit GATHER is often not required. The OUTPUT FILE component does not support this automatic fan-in, so if you want to departition data just before writing to an OUTPUT FILE, you must explicitly insert a departitioner.



Gathering the results of a data-parallel REFORMAT into a serial file

- MERGE — Reads sorted data records from multiple partitions and preserves the sort order specified by a sort key:

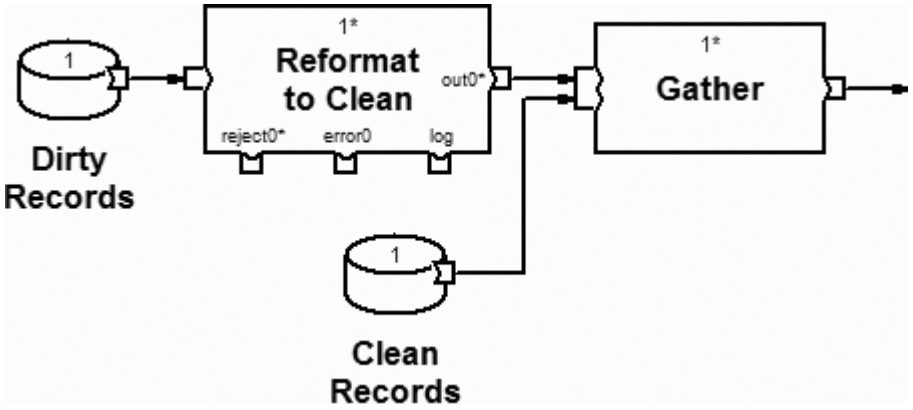


Merging the results of a sorted, data-parallel ROLLUP into a serial file

Partitioning and departitioning without data parallelism

Although the partitioning and departitioning components are most often used in a data-parallel fashion, you can also use them in a component-parallel fashion by connecting them to multiple serial flows instead of to a partitioned flow. Reread the sections on partitioning and departitioning and substitute the word “flow” for “partition.”

GATHER is particularly useful as a way of collecting records with the same format from several different components.



Using GATHER to collect records from two serial flows

Activities

Save these activities as **Parallelism_#.mp**, where **#** is the activity number.

Partitioning by round-robin

The PARTITION BY ROUND-ROBIN component distributes records to multiple partitions one at a time in sequence. Thus, if your records have a numerically sequential key and you are partitioning to a two-way multifile, the odd-numbered keys will go to one partition and the even-numbered keys will go to the other.

1. Partition the serial **Customers** dataset (**$AI_SERIAL/customers.dat**) by round-robin into a two-way multifile **mfile:$AI_MFS/Parallelism_out_1.dat**. Use **View Data** on the input file and on each partition to see how the data has been distributed. After the graph has run (or while it is running), right-click the OUTPUT FILE and choose **Tracking Details** to see the counts of records written to each partition.

Partitioning by key

Partitioning by key sends all records with the same key to the same partition.

2. Starting with the graph from the previous activity, partition the serial **Customers** dataset by **customer_id** into a second two-way multifile. Use **View Data** on each partition to see the difference between partitioning by key and by round-robin. If you set **TEST_FLAG** to **SmallData** for this activity and check the **Tracking Details** for the output multifile, note that there is a data skew of 8%. Compare that with the skew from the previous activity (data skew of 0% is noted as blank). In general, small data skew is not a problem, but as noted earlier, it can result in diminished performance under certain circumstances.

Sorting after partitioning by key

Sorting the data after partitioning by key will show you that all records with the same key are grouped together and in the same partition. While it is technically correct to sort a serial dataset before a PARTITION BY KEY, the sort will run serially, reducing performance. On the other hand, if the stream being partitioned by key is already ordered on the partition key, it will not need to be sorted again after the PARTITION BY KEY. In other words, partitioning has no effect on the sort order; you should sort only if necessary.

3. Partition the serial **Transactions** dataset (**$AI_SERIAL/transactions.dat**) by **customer_id**; then sort by **customer_id**, and write the results into a two-way multifile. Use **View Data** on each partition to see that records with the same key are grouped together in the same partition.

Gathering

The GATHER component collects records as they are available, without imposing any ordering on records coming from different partitions. A GATHER does not necessarily produce the same order of records every time, so you should not rely on the ordering of records after a GATHER component.

4. Gather the records from the multifile **Customers** dataset (**mfile:$AI_MFS/customers.dat**) into a serial file.

Merging

The MERGE component requires presorted input and preserves the order of records based on the sort key (or keys). Like the GATHER component, it can be used to combine flows.

5. Merge the records from the multifile **Customers** dataset on the **customer_id** into a serial file.

Partitioning and departitioning in parallel

Because partitioners and departitioners can run in parallel, you can efficiently repartition data in parallel. If you want to change the degree of parallelism at a particular point in a graph, you must generally use a partitioner running in the source degree of parallelism (layout), connected by an all-to-all flow to a departitioner running in the destination degree of parallelism. If the source data is presorted within each partition on the key and you want to preserve that property, use MERGE to departition the data; otherwise, use GATHER.

As a very special case, if the source degree of parallelism is an exact multiple of the destination degree of parallelism, and you are not concerned about the type of partitioning of the destination, you can connect the source directly to a departitioner with a fan-in flow.

6. Partition the serial **Transactions** file by **customer_id**, sort on **customer_id** in a four-way multifile layout (use **mfile:$AI_WIDE_MFS** as the layout), and then merge the results on **customer_id** to a two-way multifile. Note that you must specify the layout of the SORT component explicitly, because it cannot be inferred from its neighbors.

Partitioning between different degrees of parallelism while preserving sort order

To partition between different degrees of parallelism while preserving sort order, you must do another partitioning step after the SORT, and follow that step with a MERGE component. The partitioner is necessary to send the data from the source number of partitions to the destination number of partitions, and the MERGE is required to maintain the sorted order of the received data.

7. Partition the serial **Transactions** file by **customer_id**, sort on **customer_id** in a two-way multifile layout, and put the results into a four-way multifile, partitioned and sorted on **customer_id**. Note that you must explicitly specify the layout of the SORT or of the partitioner downstream of the SORT.

Another way to specify the degree of parallelism

Degree of parallelism can be specified through a multifile, but need not be. Multifile layouts are convenient, easy to use, and usually configured as some function of the number of CPUs in the system. In the next activity, you will sort in a three-way layout that has no corresponding multifile system.

▶ **To set a custom three-way layout on the SORT component:**

1. On the **Layout** tab of the SORT component's properties, click **Custom**.
2. Click **Edit**.
3. In the **Custom Layout** dialog, add partition locations (directories) by clicking **New** and then filling in the line with a serial URL. To get three copies of the component running in **file:$AI_SERIAL**, add three identical entries. If you simply press Enter after typing an entry, you will be prompted for the next one. When finished, click **OK**.

To go from the three-way layout of the SORT down to a two-way layout, you must again insert a PARTITION BY KEY component followed by a MERGE component to preserve the sort order.

8. Partition the serial **Transactions** file by **customer_id**, sort on **customer_id** in a three-way layout, and put the sorted results into a two-way multifile.

SOLUTIONS

The solution graphs are named **solutions/Parallelism_#.mp**, where **#** is the activity number.

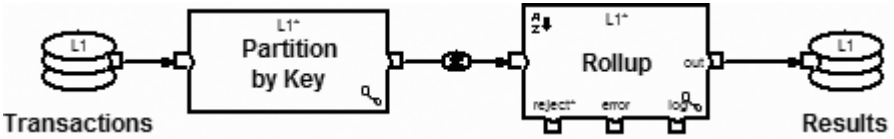
17

Repartitioning: ROLLUP and JOIN in parallel

When you rollup or join data, you specify a key to group the records by. When you do these operations in parallel, you need to make sure that all records having the same key value are together in the same partition. This is necessary because each partition is processed separately without communicating with the others. If the incoming data is parallel, you might need to *repartition* the data by the desired key before a ROLLUP, JOIN, or other key-based component.

Parallel repartitioning

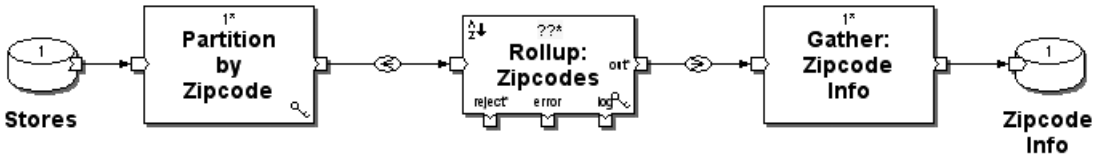
You do a parallel repartitioning of data by running a partitioner in parallel. Data will be distributed in an all-to-all fashion, potentially flowing between all combinations of upstream and downstream partitions. The built-in gather capability on most components will collect the records in each downstream partition from the multiple upstream partitions.



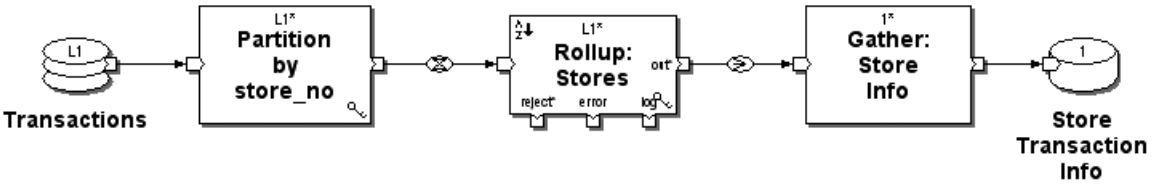
The PARTITION BY KEY component repartitions the Transactions multifile on the same key as the ROLLUP that follows.

Explicitly setting the layout when the GDE cannot determine it

In some situations, the GDE may not be able to determine the appropriate layout, so you must set it explicitly. In the following graph, for example, the intent is obviously to run the ROLLUP in parallel, but there is no parallel layout that can be propagated to it.



The graph below shows a different situation. Here the GDE has "guessed" that the ROLLUP's layout should propagate from the **Transactions** multifile, but if you wanted the ROLLUP to run in a different degree of parallelism, you would have to set its layout explicitly.



Activities

Save these activities as **Repartitioning_#.mp**, where **#** is the activity number. All solution output files should be multifiles unless otherwise stated.

First build and run each graph with the sandbox parameter **TEST_FLAG** set to **SmallData**. When you are satisfied that the graph runs correctly, change **TEST_FLAG** to L**argeData** and rerun the graph against the full data volume. Set **TEST_FLAG** back to **SmallData** before proceeding to the next activity. You can also test partitioning logic with small sample datasets you create yourself by using FILTER BY EXPRESSION to extract interesting subsets of the large data.

Repartitioning on a different key

The **Transactions** dataset is partitioned on **transaction_id**. If you want to rollup the partitioned multifile by **customer_id**, you must first repartition the data on the **customer_id** to make sure that all records with the same **customer_id** are in the same partition.

1. Using the multifile **Transactions** dataset as input, count the number of transactions made by each **customer_id**.

Partitioning on the key needed for a join

The **Customers** dataset is partitioned on the **customer_id**, while the **Transactions** dataset is partitioned on the **transaction_id**. To join these two datasets, you must first partition them on the key you intend to join on.

2. Produce a multifile dataset that attaches to each transaction record a field named **customer_name**, consisting of the customer's first name and last name separated by a space. Send customer records for which there are no transactions to a TRASH component.

   The following activity again requires partitioning on the appropriate key before joining.

3. A customer who has not made any transactions in a given period is considered "inactive." Find all inactive customers for the year 1999.

Rolling up within partitions and then across partitions

Sometimes you do not have to partition before rolling up. You can first rollup within partitions and then use a second ROLLUP component to rollup across partitions. This situation often occurs when you want to summarize an entire dataset to a single record. It can also be useful for cases where there would be large data skew after repartitioning.

4. Find the total spent across all stores in each year by first rolling up transactions within each partition and then summing the results across partitions. The first ROLLUP should run in parallel — you might have to set its layout explicitly, as described on page 156 — and the second should run serially, with the layout of the serial output file.

SOLUTIONS

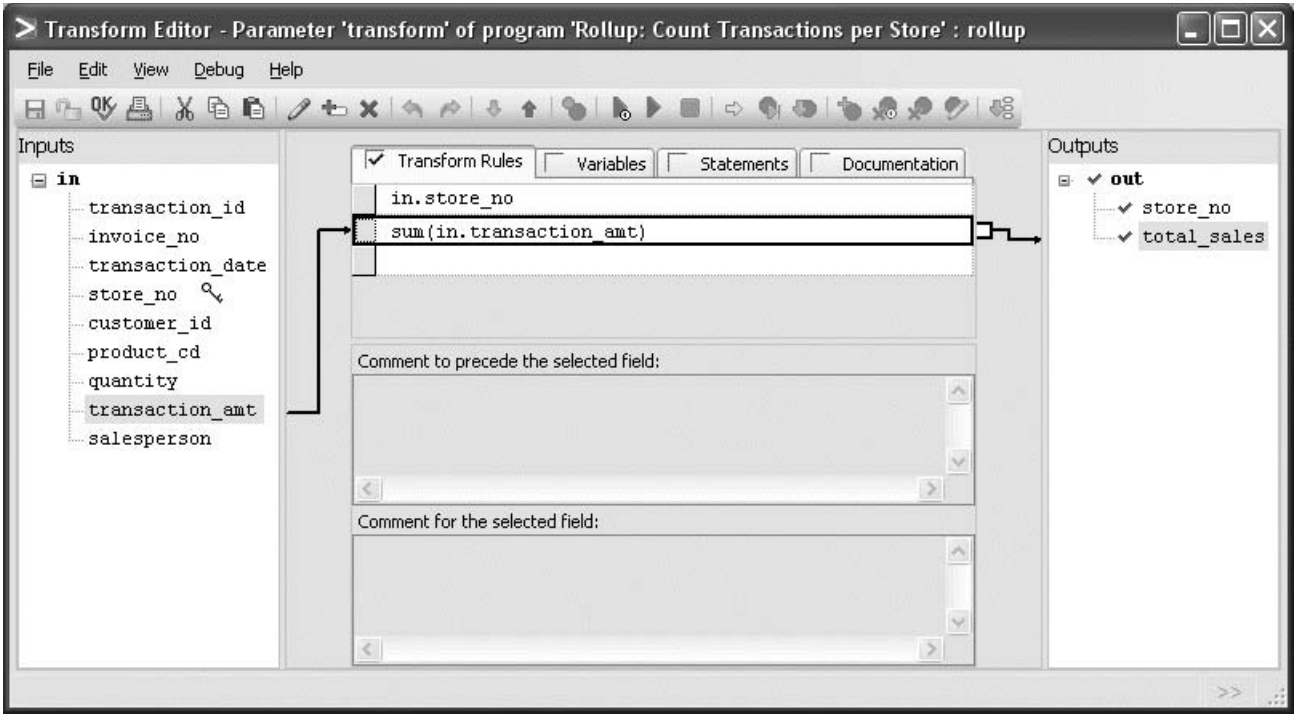
The solution graphs are named **solutions/Repartitioning_#.mp**, where **#** is the activity number.

18

Multistage transforms: ROLLUP without aggregation functions

Most components that can transform data, such as REFORMAT and JOIN, are controlled by a single transform function. This transform function is executed once for each input record (as with a REFORMAT), or once for each set of matching input records (as with a JOIN). Components that process their records in groups require a richer set of transform functions. These components are called *multistage* components, because different transform functions are used at different stages during their execution. ROLLUP is one such component. When you use its simplified interface to write a single transform function using built-in aggregation functions like **count** and **sum** (as you did in the activities in Chapter 14), that transform function is expanded into a series of multistage functions at runtime.

The following figure shows (in grid view) a transform using the **sum** aggregation function to get the total sales for each store from the **Transactions** dataset:



Using the Package Editor

The built-in aggregation functions are sufficient for the most commonly computed types of aggregations and can be combined with other functions to compute more complex expressions. However, in some rare circumstances, even combinations of the built-in aggregation functions with other functions are not sufficient to solve the problem. In those cases, you can directly edit the multistage transform functions in the ROLLUP component by using the **Package Editor**.

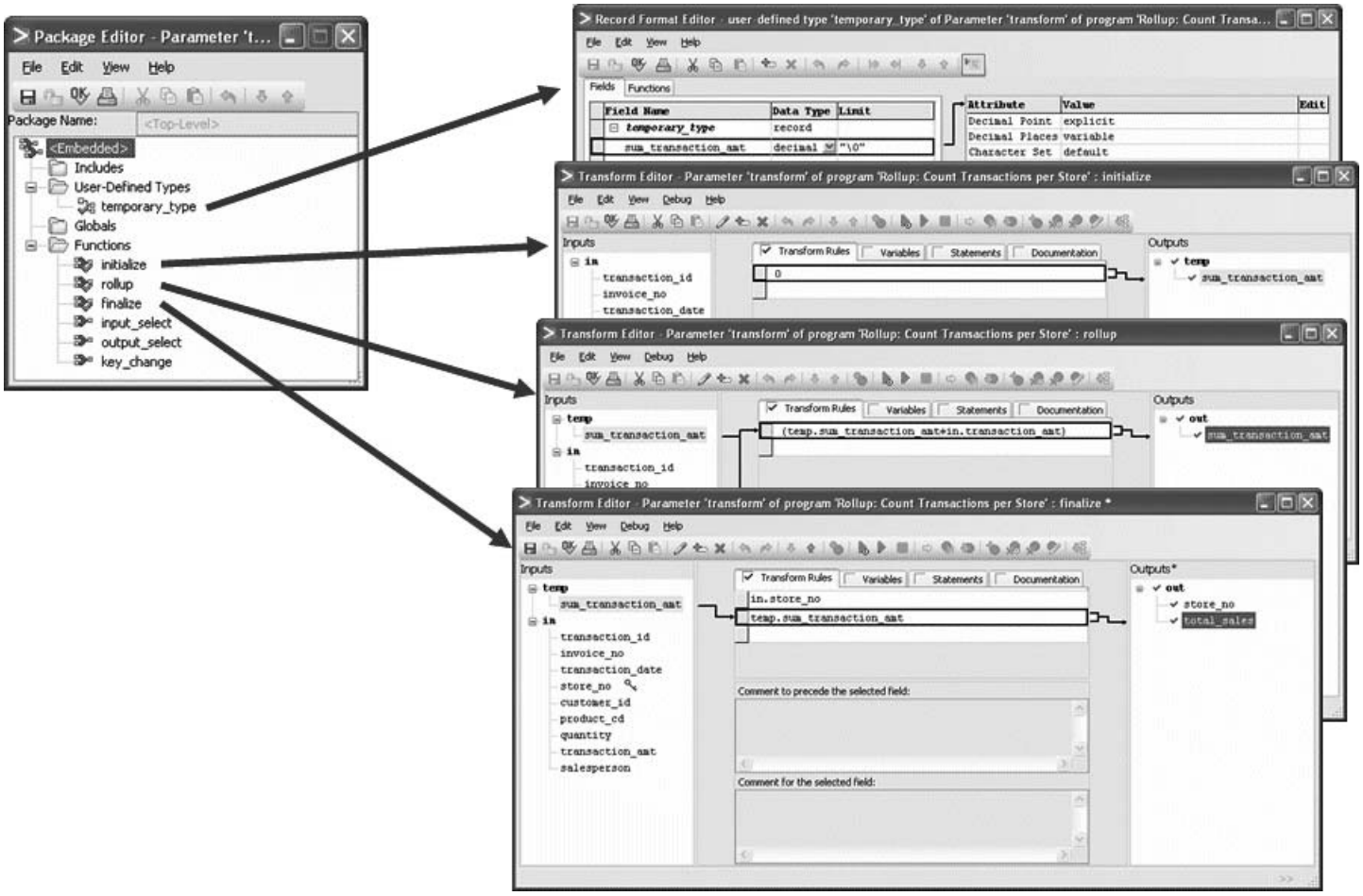
▶ **To open the Package Editor for a multistage ROLLUP:**

1. Shift + double-click the ROLLUP to open the **Transform Editor**.
2. If necessary, choose **View > Grid View**.
3. Choose **Edit > Expand Rollup** and click **Yes** in the dialog that asks whether you want to continue.

   The **Transform Editor** closes.
4. Shift + double-click the ROLLUP again.

   The **Package Editor** appears. The **temporary_type** and all functions are editable.

In the **Package Editor**, you can edit each function in the multistage transform by double-clicking the function's name. The functions in a multistage transform are a template: the body of each function can be customized for different computations, but the name of the function must remain the same. The next figure shows the **Package View** of the multistage transform that corresponds to the **sum** calculation (total sales for each store) shown earlier:



Package View displays the names of the available functions for a component's multistage transform. Not all functions are required; some, like **input_select** and **key_change**, are optional. This chapter discusses only the required functions for ROLLUP. For more information about the optional functions, see the *Component Reference*.

The functions in a multistage transform

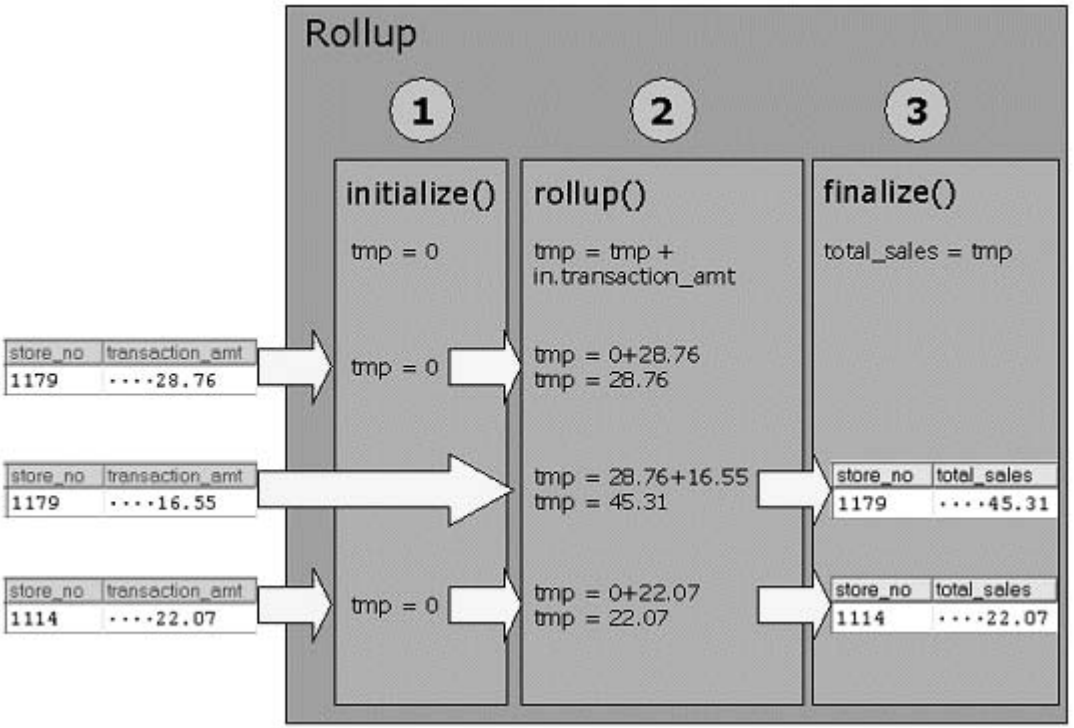
During the ROLLUP's execution, each function in the multistage transform is called in a specific order depending on whether the incoming record is the first or last in the key group, or in the middle. The **temporary_type** data type is used to define variables that will hold the results of the computation between input records and also between calls to the different functions in the transform.

The functions in the multistage transform are called in the following order:

1. **initialize** function — Assigns starting values to the variables in the **temporary_type**.
2. **rollup** function — Is an "iterator" function. The **rollup** function resulting from expanding the rollup is called once for each input record and is used to update the values of the variables in the **temporary_type**. (In a ROLLUP with a single transform, this is done automatically at runtime when the transform is expanded, as described on page 159.)

   Each multistage transform component's iterator function has a different name; for example, the SCAN component's iterator function is named **scan**.
3. **finalize** function — Creates an output record.

The following diagram shows how the multistage functions would be executed to find the total sales for each store in the DIY company.



The key of the ROLLUP component is the **store_no** field. When the ROLLUP gets the first record, it calls the **initialize** function to initialize the **tmp** variable to **0**. That record is then sent to the **rollup** function, which updates the value of **tmp** to be the sum of its original value (zero) and the **transaction_amt** from the record (**28.76**). Then the next record is sent to the ROLLUP component. This record has the same key value for **store_no** (**1179**), so the **initialize** function is not executed. Instead, this record is sent directly to the **rollup** function, which updates the value of **tmp** again with the **transaction_amt** from the current data (**16.55**). At this point, there are no remaining records with the same **store_no**, so the **finalize** function is called to produce an output record for the group of records whose **store_no** is **1179**. The total sales is reported as the value of the **tmp** variable.

When the ROLLUP component gets the next record, it calls the **initialize** function to reset the value of the **tmp** variable back to **0**. The record is then sent to the **rollup** function, which updates the value of the **tmp** variable. Because this is the last record in the dataset, the **finalize** function is then executed to output the final aggregate record.

To summarize:

- The **initialize** function is executed once for the first record encountered in each key group.
- The **rollup** function is executed once for each input record, regardless of key value.
- The **finalize** function is called once for each key group after the last record in that group has been processed by the ROLLUP. It is used to produce an output (aggregate) record for the previous key value.

Defining a multistage transform

▶ **To define a multistage transform:**

1. Define the **temporary_type**. Usually this will be a record type. Create one field for each quantity you need to calculate. For example, if you were calculating an average, you would need two fields: one for the sum of the data, and another for the count of records seen so far. For each field, choose a data type that is appropriate for the type of data being computed and is large enough to hold the aggregated values of the data.

   > **NOTE:** The **temporary_type** should include only values that will change as the ROLLUP is executing. Thus, in the example above that calculates the total sales for each store, you do not need to include **store_no** in **temporary_type**. Instead, simply use the **finalize** function to access it from the input record and output it.

2. Define the **initialize** function. For numeric data, the temporary fields are usually initialized to **0**; for string data, an empty string (**""**) is typically used.

3. Define the **rollup** function. Decide how you want to incrementally compute your result by updating the value of the fields of the **temporary_type**. Here are some common examples:

TO COMPUTE	WRITE AN EXPRESSION LIKE THIS
sum	`tmp + in.value`
count	`tmp + 1`
max	`if (in.value > tmp) in.value` `else tmp`
min	`if (in.value < tmp) in.value` `else tmp`
average	Compute a sum and a count in the **rollup** function; use the **finalize** function to divide the sum by the count.

4. Define the **finalize** function. Decide what information you want to put into the output records for the component. You will typically use the variables in the **temporary_type** as part of your output record. For example, when calculating a sum, you will likely output the sum computed in the **temporary_type** as the sum in the **finalize** function.

Activities

Save these activities as **Rollup_multi_#.mp**, where **#** is the activity number. Use the datasets in **$AI_SERIAL**.

Keeping track of values associated with minimums or maximums

The ROLLUP component's aggregation functions cannot be used to solve problems like this: "Find the largest transaction made in each store. Include the **customer_id** of the customer who made this transaction." The key for the ROLLUP is the **store_no**, and an aggregation function (**max**) is applied to the **transaction_amt** field. The **customer_id** field is neither part of the key nor aggregated. In this case, the ROLLUP component will return either the first or the last **customer_id** for each **store_no** (depending on whether the ROLLUP is in-memory or sorted). That **customer_id** will probably not correspond to the record that contains the largest **transaction_amt** for that **store_no**.

One way to solve a problem like the one above is to edit the multistage transform in the ROLLUP component to keep the **customer_id** that corresponds to the maximum **transaction_amt**. The simplest way to begin is to use the ROLLUP aggregation functions as usual to compute the maximum **transaction_amt** for each **store_no**. Then edit the multistage transform (using the **Package Editor**) to add logic to keep the corresponding **customer_id**. You will need to add a new variable to the **temporary_type** (to store the correct **customer_id**). You will also need to edit the **rollup** function to update the **customer_id** as necessary. (Hint: Use the same logic the function uses to keep track of the largest **transaction_amt** seen so far.) Finally, update the **finalize** function to return the **customer_id** you stored.

1. Find the largest purchase made in each store. Include the **customer_id** of the customer who made the purchase.

Using a simple type for **temporary_type**

Sometimes you want an aggregation function to keep track of only one value. To do this, edit the **temporary_type** field to change its data type from **record** to the DML data type you need. For example, changing the **temporary_type** to a one-character string would look like this in the **Record Format Editor**:

	Field Name	Data Type	Limit
	temporary_type	string	1

2. Create a list of customers that contains the **customer_id** and a field called **meets_cutoff** that indicates whether the customer has ever made a transaction greater than $100. The **meets_cutoff** field can have one of two values: **"Y"** if the customer has made a transaction greater than $100 and **"N"** if not.

Using a record type for **temporary_type**

When you need an aggregation function to keep track of more than one value, you can define **temporary_type** as a record type.

3. Find the smallest purchase made in each store. Include the **customer_id** of the customer who made the purchase and the date the purchase was made.

Combining built-in aggregation functions with other functions

You can write complex expressions by combining the built-in aggregation functions with other functions. In such cases, you don't need to edit the multistage transform; you can simply use the standard ROLLUP transform. Suppose you want to calculate the number of days that elapsed between a customer's earliest transaction and most recent transaction. You would write an expression like this:

```
date_difference_days(max(in.transaction_date),
                     min(in.transaction_date))
```

4. Calculate how many days have elapsed since each customer's most recent purchase. Include the **customer_id** and the number of days since the last purchase in your solution. Do not use the multistage transform in the ROLLUP component to calculate your results.

SOLUTIONS

The solution graphs are named **solutions/Rollup_multi_#.mp**, where **#** is the activity number.

19

Accumulating across groups of records: SCAN

The SCAN component generates, for every input record, an output record that includes a running cumulative summary for the group the input record belongs to. Two common uses of SCAN are for computing the amount spent to date or for assigning unique numbers to each record in a group. SCAN has much in common with ROLLUP, as the following table shows:

FEATURE	SCAN	ROLLUP
Outputs intermediate computation results (one for each record)	✓	
Outputs the final aggregated results of its processing	✓	✓
Produces a single output record for each input record (like REFORMAT)	✓	
Saves information from previous records and applies it to the current calculation for each record (unlike REFORMAT)	✓	✓
Groups its records by a common key value	✓	✓
Has built-in aggregation functions that do not have to be written by hand		✓
Can use a multistage transform	✓	✓
Has a **sorted-input** parameter that specifies whether the data must be presorted or whether the component will group the data in memory	✓	✓

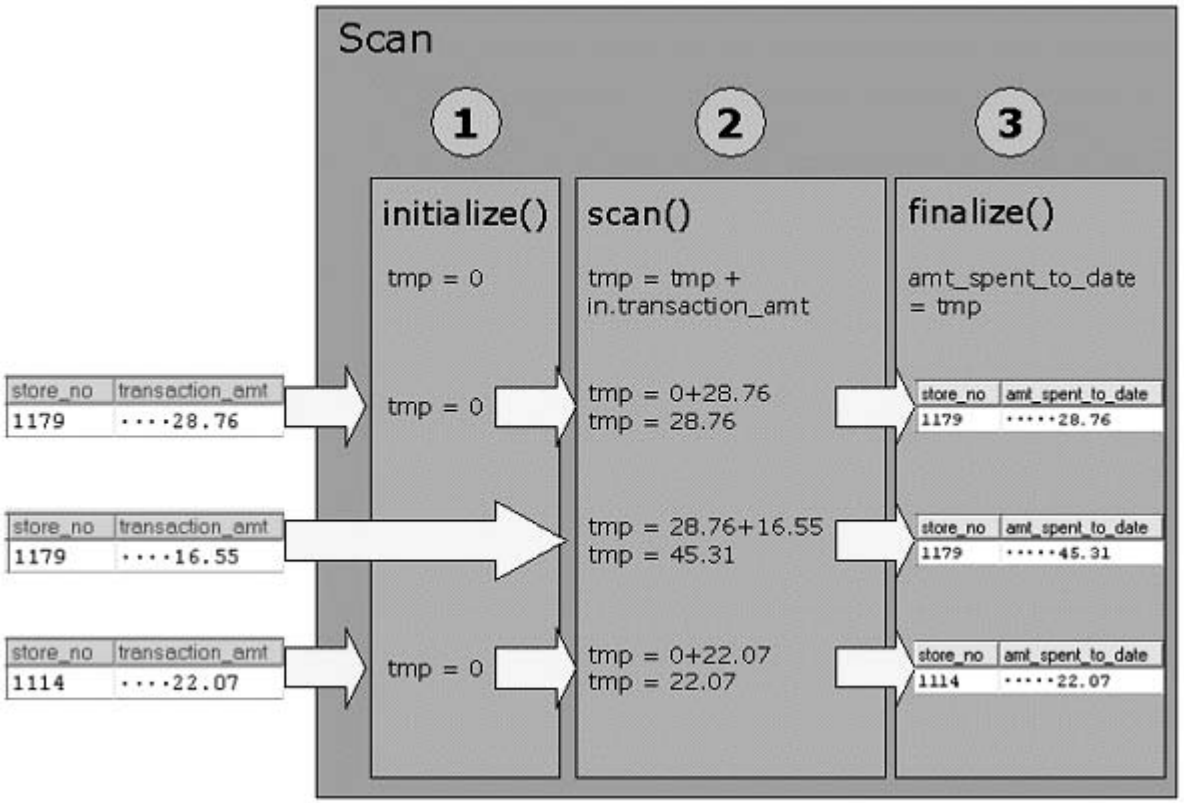
Similarities between multistage transforms for ROLLUP and SCAN

The multistage transforms for the ROLLUP and SCAN components are very similar. The multistage transform in a ROLLUP component that computes the **sum** of a field, or a SCAN component that computes a running total, would contain the following:

- A variable in the **temporary_type** to store the **sum** calculated for that field
- An **initialize** function that resets the **temporary_type** sum to **0**
- A function (named **rollup** in a ROLLUP component, or **scan** in a SCAN component) that adds the **temporary_type** sum to the amount in the input record

- A **finalize** function that outputs the value of the **temporary_type** sum as the aggregate sum for each group of records

  In the ROLLUP component, **finalize** is executed once after all the records in a group have been processed; in the SCAN component, **finalize** is executed once for each input record after that record is processed. Thus, the ROLLUP component uses **finalize** to produce a single aggregate record for each group; and the SCAN component uses **finalize** to produce one cumulative record for each input record.



Activities

Save these activities as **Scan_#.mp**, where **#** is the activity number. Use the datasets in **$AI_SERIAL**.

1. Add a new field, **transaction_number**, to the **Transactions** dataset. The **transaction_number** should be a sequential number (starting at **1**) that uniquely identifies each transaction made

by a specific customer (**customer_id**). Each **customer_id** will have one **transaction_number** equal to **1**, but no **transaction_number** should be repeated within the same **customer_id**.

Sorted versus in-memory

The SCAN component, like the ROLLUP component, can be used with sorted data or can store its computations in memory. For some computations, it is necessary to presort the data by a more detailed key than the SCAN component uses. For example, to ensure that results reported by the SCAN component for **store_no** are accumulated in calendar order, you should sort the data by **{store_no; transaction_date}**, even though the SCAN component uses only **{store_no}** as its key.

2. Add a new field, **transaction_number**, to the **Transactions** dataset. The **transaction_number** should be a sequential number (starting at **1**) that uniquely identifies each transaction made in each store (**store_no**). Assign the lowest **transaction_number** to the first transaction made in each store, and the highest **transaction_number** to the most recent transaction made in each store. If two transactions from the same store have the same transaction date, the one with the lower **transaction_id** should come first.
3. Compute the cumulative sales made per day in each store. Include the **store_no**, **transaction_date**, and **sales_to_date** in your result. Make sure that your solution contains only a single record (the last record) in cases where multiple transactions occurred on the same day.

Using the empty key to treat all records as a single group

Like the ROLLUP component, a SCAN can use the empty key: **{}** to treat all records in a dataset as a single group. You can use this technique to add a field that uniquely identifies records that are not sorted by a specific key, and you can later use that field to restore their original order if it is lost through re-sorting or through the use of a component like GATHER.

4. Create a new transactions dataset that does not include invalid transaction dates or amounts. The new transactions dataset should be partitioned on the **customer_id** and should have only the fields that were in the original serial **Transactions** dataset. Collect all records with invalid transaction dates and amounts in a serial file. The invalid records should contain a field that indicates their record number in the original serial **Transactions** dataset, in addition to all fields in that dataset. (For example, if the fifth record in the original serial **Transactions** dataset is invalid, this record should have a **record_no** = **5**.) The invalid records should also be in the same relative order as in the original dataset (sorted by **record_no**).

SOLUTIONS

The solution graphs are named **solutions/Scan_#.mp**, where **#** is the activity number.

20

Business problems

Having completed all the activities in *Work>Book 1*, you can now apply what you've learned to a more sophisticated set of business problems. These problems are designed to address issues that typically occur in everyday business and to demonstrate how you can solve complex problems by first reducing them to collections of simpler ones. Each of the following problems can be broken down into a set of questions of the kind you have already solved.

Each problem in the first group is immediately followed by a breakdown into simple steps, and then a list of details to read if you are stuck. For the second group, the breakdowns and details are provided in the Appendix (page 193); you should try to solve the problems without them. For the final group of problems, no breakdown or details are provided; you are on your own.

Use multifiles for all solution output files unless otherwise instructed.

Top-down problem solving, bottom-up construction

A powerful and natural approach to implementing complex business processes in Ab Initio graphs is to design from the top down. Start with the process as a whole and identify the major independent subprocesses that compose it. Whether your process will ultimately consist of one graph or several depends on its complexity, but this is not important at the design stage. Just identify which fields need to pass from one subprocess to another; these will become the data in flows between components or subgraphs, or the data in files between graphs. You will later associate fields to specific datasets or flows, based on the details of how they are produced by the subprocess.

Next, consider each subprocess in turn and again identify the major steps involved. Keep repeating this procedure on the major steps until you reach subprocesses you can assemble in a relatively straightforward manner using Ab Initio components. You can then start building from the bottom up, implementing and unit-testing one subprocess after another, and connecting and integration-testing them as they are completed.

As you work from the details to the larger pieces, you may notice improvements that you can make to the overall structure. For example, you might realize that the same calculation is being done in multiple places, and that you could eliminate the redundant calculations by changing the design. This is a natural part of building applications with Ab Initio software: because both the details of the implementation and the overall architecture can be viewed at once, they can influence one another in beneficial ways.

You will achieve the biggest gains in performance tuning by re-architecting the graph to obey these guidelines. Low-level tuning tends to produce small improvements, not big ones. The GDE makes it easy to re-engineer applications quickly, and you should take full advantage of this capability.

You should solve each of the following business problems with a graph that runs in parallel. It is good practice to develop your graphs using the **SmallData** datasets before running with **TEST_FLAG** set to **LargeData**, though this may not be possible for all problems.

BEST PRACTICE

Design with performance in mind.

Two important performance guidelines that you should consider as you break down your problems are:

1. Minimize the number of times that your graph reads a given dataset.
2. Reduce the data being carried through the graph to only the fields and records used in the process or needed in the output.

Business problems with breakdown and details

This section presents the following business problems:

The solution graphs are named **solutions/BusinessPb_#.mp**, where **#** is the problem number.

01. Supply and demand

THE PROBLEM

All products in the home improvement department have a wholesale price, but no item is sold at that price, and the DIY company allows every store to set its own markup for each item. The DIY company wants to determine whether the customers who pay the highest average markup are among the best customers in terms of total amount spent. The markup for a product is defined as the total amount paid minus the wholesale price. Find the customers who paid the biggest markup on average across all their purchases. The output record should contain the customer's ID, the average markup paid, and the total amount spent. Sort the output in descending order by average markup paid.

THE BREAKDOWN

1. Roll up the **Transactions** dataset to find the average markup paid per transaction for each customer and the total amount spent.
2. Sort by the average markup paid.

THE DETAILS

1. Use the **Products** dataset as a lookup file.
2. Roll up on the **customer_id** and use the **avg** aggregation function to find the average markup paid per transaction and the **sum** function to find the total amount spent. Use lookup functions to get the wholesale price from the **Products** lookup file. Drop any unnecessary fields in the ROLLUP; keep only the customer ID, average markup paid, and total amount spent.
3. Some transaction records have blank transaction amounts. These records will be rejected by the ROLLUP component. Collect them in an output file.
4. Sort by **{avg_markup descending}**.

The solution graph is **solutions/BusinessPb_01.mp**.

02. Best sellers

THE PROBLEM

The DIY company would like to know which products were its best sellers in November 2000. Find the total number of units of each product sold, the average sales price per unit of the product, and the total sales for the product. The output should be ordered by the total sales of the products. Also include the product's name, and the wholesale price for comparison. Transactions with blank transaction amounts should not be included, but instead can be ignored. Compute in parallel but produce a serial result dataset.

THE BREAKDOWN

1. Find the transactions made in November 2000. (Be wary of blank dates.)
2. Separate transactions with a blank transaction amount into a data quality file.

3. Roll up the selected transactions to find the total number of units sold per product, the average amount spent on the product over all transactions, and the total amount spent on the product.
4. Attach the product name and wholesale price information to the transactions.
5. Order the results by the total sales per product.

THE DETAILS

1. Partition the **Transactions** dataset so that you will be able to compute the products with the biggest sales.
2. In a FILTER BY EXPRESSION component, use the built-in **date_month** and **date_year** functions to select only the transactions made in November 2000.
3. Use a FILTER BY EXPRESSION component to select transactions with nonblank transaction amounts. Collect the deselected records in a data quality file.
4. Use an in-memory ROLLUP. Edit the output record format to include the fields **product_cd**, **product_name**, **units_sold**, **avg_price**, **total_sales**, and **whs_price**. For the four numeric fields, use the types **integer(4)**, **decimal(9.2)**, **decimal(12.2)**, and **decimal(9.2)**, respectively. Remove all other fields except the **product_cd** and the **product_name**. Roll up on the **product_cd** key, using the **sum** and **avg** aggregation functions to find the necessary summary information. Using the **Products** file as a lookup, call lookup functions to get the product name and wholesale price.
5. Sort the final results on the total sales per product (**total_sales**) in descending order.
6. Merge the sorted results on **{total_sales descending}** into a serial file.

The solution graph is **solutions/BusinessPb_02.mp**.

03. Promoting regional managers

THE PROBLEM

The DIY company has stores in five geographic regions. Each of its salespeople is assigned to customers from all five regions. Whenever a customer makes a purchase in any region, the assigned salesperson is credited with the sale. All of DIY's store managers are also salespeople (but not all salespeople are store managers).

To help coordinate changes to the home improvement departments in its stores, DIY is considering appointing regional managers: if any current store manager is also a top salesperson (in any region), the company will promote that manager to one of these new regional manager positions.

Identify the top salesperson (the salesperson with the largest total sum of **transaction_amts**) in each region. Indicate whether the top salesperson is a store manager, and if so, include the store number and the city and state where the store is located.

THE BREAKDOWN

1. Look up the region code for each transaction. Drop any unneeded fields.
2. Roll up the transactions to find the total sales for each salesperson in each region.
3. Find the salesperson in each region with the largest total transaction amount.
4. Look up to see whether each salesperson is a manager, and attach the city and state of his or her store.

THE DETAILS

1. Partition the **Transactions** dataset so that you will be able to compute the total transaction amount for each salesperson.
2. The region codes are stored in the **Regions** dataset, which you should access as a lookup file. Use lookup functions in a REFORMAT to add a field for the **region_code**. Drop from the output record format the fields you will not need (**transaction_id**, **invoice_no**, **transaction_date**, **store_no**, **customer_id**, **product_cd**, and **quantity**).

3. Roll up on the **salesperson** and **region_code** fields. Use the **sum** aggregation function to total the **transaction_amt**. Ignore the blank transaction amounts by using the condition **is_valid(in.transaction_amt)** in the second argument of **sum**.
4. Partition the output of the ROLLUP component to group the salespeople by region.
5. Use a SORT component followed by a DEDUP SORTED component to identify the salesperson with the largest total transaction amount in each region. The SORT should be on the compound key **{region_code, total_trx_amt}**. If the sort order for the **total_trx_amt** field is set to ascending (the default), DEDUP SORTED should be set to take the last record in each group (with the key **region_code**). Otherwise (if the **total_trx_amt** is sorted in descending order), the DEDUP SORTED should be set to take the first record (the default).
6. Use the **Stores** dataset as a lookup file to get the manager information and store locations. In a REFORMAT, use lookup functions to set a flag indicating whether a salesperson is a manager, and to attach the **store_no**, **city**, and **state** of the store he or she manages.

The solution graph is **solutions/BusinessPb_03.mp**.

04. The January sale

THE PROBLEM

It is the beginning of the first quarter, and sales of home improvement items are expected to be low after the holidays. To boost sales, the DIY company has decided to put five items on sale. Find the best-selling item and the four slowest-selling items in December 2000 by total sales after applying the following criteria:

- Ignore as seasonal all products that sold more than four times as much in the previous June as they did in December.
- Consider only items for which total December sales were greater than $250.

The final output should include only the following fields: product name, wholesale price, total sales in December 2000, and average amount paid for the item in December. List the selected products in decreasing order of total sales.

THE BREAKDOWN

1. Eliminate all transactions except those from June and December 2000.
2. Within the two periods, calculate each product's total amount and total quantity sold. At this point you have a series of product records, sorted into two groups by product code and month.
3. Split the product records into two groups, one for June and one for December. Compare the groups and figure out which products had seasonal sales, and what the products' average sales were. Keep the data only from the December sales.
4. Eliminate the seasonal products. The remaining stream of December records, when sorted according to **total_amt**, will contain at its high end the single best-selling product, and at its low end the four lowest-selling products.
5. Extract the five records and put them in the output file, at the same time adding to each record the product name and **whs_price** information from the **Products** dataset.

THE DETAILS

1. You don't need the product names and wholesale price information (from the **Products** dataset) to do the calculations for this problem, so you should wait until the end to add this information to the output records.
2. Filter out all transactions except the ones from June and December 2000. At the same time, eliminate invalid dates. You can reformat to get rid of unneeded record fields here, but you need to keep the month/year date information, because you will need to compare the product records on this basis later on.
3. Roll up to get the transaction amounts (total sales) and quantities sold per product.

4. At this point you have a flow of product records sorted in two month/year groups. Filter these into two flows, one consisting of the product transactions for June 2000 and the other those for December 2000. Then use a JOIN to compare the two flows: in the transform you can calculate the average sale price for December, as well as figure out whether a product's June sales were seasonal. Save this information as the ratio of June sales to December sales.
5. Keep only the sales data for nonseasonal products (those for which the June-to-December sales ratio was less than or equal to four) and those for which total December sales were greater than $250.
6. Merge the partitions into a single serial flow (since the output dataset will be a serial file, consisting of only five records).
7. Sort the records on **total_sales** (ascending).
8. Filter the first four records of this flow (using the built-in **next_in_sequence** function); this gives you the four product records with the lowest amounts. At the same time, use a DEDUP SORTED component on the deselected records to get the last record (which, because of the ascending sort, will be the record with the highest **total_amt**). Do this by specifying **{}** as the key for the DEDUP SORTED component, and **last** for the **keep** parameter.
9. Use a GATHER component to combine the two flows, and a SORT to put the five records in order of descending **total_amt**.
10. Reformat the records into their final form, and look up and add each product's name and **whs_price** from the **Products** dataset.
11. Sort by **total_sales** (descending).

The solution graph is **solutions/BusinessPb_04.mp**.

05. Cumulative sales per store

THE PROBLEM

The DIY company wants to see the cumulative sales for each of its stores. Each output record should contain the store number (**store_no**), year (**transaction_year**), and cumulative sales (**cumulative_sales**). Collect transactions with invalid dates or amounts in a reject file for later analysis. Optionally, learn about the VALIDATE RECORDS component (in the **Validate** folder in the **Component Organizer**) in Ab Initio Help and use that to collect the invalid records.

THE BREAKDOWN

1. Create a field that contains the year in which each transaction was made.
2. Sort the data by the **store_no** and **year** (in ascending order).
3. Roll up the data to compute the total sales in each store per year.
4. Use a SCAN component to calculate the cumulative sales over the years in which transactions were made for each store.

THE DETAILS

1. Partition the **Transactions** dataset so that you will be able to compute the sales per year for each store.
2. Filter out transactions with invalid dates or amounts.
3. Use a REFORMAT component and the **date_year** function to compute the year in which each transaction was made. Create a new field, **transaction_year**, and output the result of the **date_year** function to this field.
4. Sort by **{store_no; transaction_year}** ascending.
5. Roll up using **{store_no; transaction_year}** as the key. Compute the **sum** of **transaction_amt** in the ROLLUP; assign this value to a new field named **sum_transactions**.
6. Set the key of the SCAN to **{store_no}**. In the SCAN transform, create a **temporary_type** with a single variable called **sales**. The **initialize** function should set the value of **sales** to **0**. The **scan**

function should compute cumulative sales by adding the **sum_transactions** value from the input record to the sales value from the **temporary_type**. The **finalize** function should output the **store_no**, the **transaction_year**, and the value of the **sales** variable as the **cumulative_sales**.

The solution graph is **solutions/BusinessPb_05.mp**.

Business problems without breakdowns

For the problems in this section, you should break the problem down on your own before solving it. If you have trouble, see the suggested breakdowns in the Appendix (page 193). The problems are as follows:

- 06. Frequent buyers (next)
- 07. Updating the product list (page 188)
- 08. Compliance with inventory clearance (page 189)
- 09. Out-of-town customers (page 189)

06. Frequent buyers

THE PROBLEM

How many different products does a customer usually buy? Find the average number of different products that each customer purchases in a single day. If the same customer purchases the same product more than once in a single day, count only one purchase.

The solution graph is **solutions/BusinessPb_06. mp**.

07. Updating the product list

THE PROBLEM

The product list for the home improvement department needs to be updated for the start of the new fiscal year, January 1, 2001. The marketing group has produced a list of new products, discontinued products, and products with proposed wholesale price changes. These are in **$AI_SERIAL/products_mkt_prop.dat**, which has an **update** field with three possible values: **N** (for new products), **D** (for discontinued), and **P** (for price updates).

Update the product list using the proposed changes, applying the business rules below. Write the new and updated records to a new products file.

1. Add each new product and set its **last_modified_date** to the date of the new fiscal year (January 1, 2001).
2. Remove all discontinued products and collect them in a discontinued file. Add a field, **discontinued_date**, and set it to the date of the new fiscal year.
3. If the wholesale price was last modified before July 1, 2000, replace it with the proposed price change, and set the **last_modified_date** to the date of the new fiscal year; otherwise, keep the existing price.
4. Collect the unchanged product data, the new products, and the products whose prices have been changed into a single file. This will be the new products dataset.

The solution graph is **solutions/BusinessPb_07.mp**.

08. Compliance with inventory clearance

THE PROBLEM

In the fourth quarter of 1999, the DIY company ordered all its stores to clear as much of their inventory of 1/16-inch bolts as possible. They would now like to know which home improvement departments complied with this directive most effectively in terms of total sales and quantity sold.

Find the five stores that made the most sales selling the product named **1/16 in. Bolt** in the fourth quarter of 1999 (October, November, and December), and the five stores that sold the largest quantity. In the final output, include the **store_no**, product code, year, quarter, total amount spent, number of units sold, and the city and state where each store is located. Use the MFS (not serial) version of the **Transactions** dataset. Save the five stores with the highest total sales into one file, and the five stores with highest total quantity into a second file.

> **NOTE:** If you run this graph with **TEST_FLAG** set to **SmallData**, you will not get any output. Instead, you may want to create a dataset of your own containing only transactions on 1/16-inch bolts.

The solution graph is **solutions/BusinessPb_08.mp**.

09. Out-of-town customers

THE PROBLEM

The DIY company is considering standardizing its prices across stores in different towns. This will require a corporate effort to coordinate advertising, rather than leaving it to the individual stores to work with the local media. To decide whether this is worthwhile, the company would like to know how many customers shop in stores outside the town in which they live and what percentage contribution the out-of-town customers make to total store purchases. Produce a listing by store

number containing the number of customers from out of town, the total number of customers, the amount spent by out-of-town customers, the total amount spent, and the percentage of the total spent by out-of-town customers.

The solution graph is **solutions/BusinessPb_09.mp**.

Final business problems

Solve each of the following problems by breaking it down yourself.

10. Tracking the sales staff

THE PROBLEM

How many salespeople are active in each store? Produce a dataset that contains all the fields from the **Stores** dataset, along with the number of salespeople who have sold products in each store.

The solution graph is **solutions/BusinessPb_10.mp**.

11. Ranking the sales staff by store

THE PROBLEM

Which stores have the most effective salespeople? Rank the stores by totaling the transactions for each salesperson in each store and then computing the average per salesperson in that store. Produce a file of store records with this rank and the average attached for the stores with ranks between 26 and 50.

The solution graph is **solutions/BusinessPb_11.mp**.

12. Comparing sales rank to level of discounts

THE PROBLEM

The DIY company is concerned that the total sales for the home improvement departments in some of its stores are very low. Before considering ways to boost business in these stores, they would like to understand how discounts have affected total sales. They would like to compare each store's sales rank (starting with #1 for the store with the most total sales) with the markup (average percentage of the wholesale price) paid by customers at that store.

Create a report that contains the store number, total sales, sales rank, and average markup for all stores. Order the data in ascending order by markup.

The solution graph is **solutions/BusinessPb_12.mp**.

13. Ranking the sales staff across the country

THE PROBLEM

Create an **Employee** file consisting of the following information for each salesperson:

- **store_no**
- Earliest transaction date associated with that salesperson
- Most recent transaction date associated with that salesperson
- Average transaction total per day
- Average number of sales per day

Sort the file in descending order based on the average daily transaction total.

The solution graph is **solutions/BusinessPb_13.mp**.

14. Cumulative sales and transactions

THE PROBLEM

The DIY company wants to see the cumulative amount spent by each customer per month. Each output record should contain the **customer_id**, **transaction_year**, **transaction_month**, and cumulative amount spent through that month (**cumulative_amount**).

The solution graph is **solutions/BusinessPb_14.mp**.

APPENDIX

Breakdowns for business problems in Chapter 20

This section breaks down each of the problems detailed in "Business problems without breakdowns" starting on page 187 in Chapter 20.

06. Frequent buyers

THE BREAKDOWN

1. Dedup the **Transactions** dataset by **{customer_id; transaction_date; product_code}** to remove duplicate purchases of the same product on the same day.
2. Roll up on **{customer_id; transaction_date}** to count how many different products each customer purchased each day.
3. Roll up on **{customer_id}** and use the **avg** function to compute the average number of different products purchased each day a customer shops.

07. Updating the product list

THE BREAKDOWN

1. Use a JOIN component as follows to difference the existing **Products** file with the proposed changes supplied by the marketing group:
    a. Make the JOIN explicit, with record required for the proposed changes file.
    b. Use prioritized rules to determine whether the wholesale price should be modified, and to set the **last_modified_date** appropriately on new or modified records.
2. Reformat to update or keep the wholesale price and **last_modified_date** as needed.
3. Filter and reformat the discontinued products and the updates.
4. Gather the updates and unchanged products.

08. Compliance with inventory clearance

THE BREAKDOWN

1. Find all transactions for the product with the name **1/16 in. Bolt** in the fourth quarter of 1999.
2. Roll up these transactions to find total sales and the number of units sold per store, attaching the city and state information for each store.
3. Find the five stores with the largest total sales.
4. Find the five stores that sold the largest quantity.

09. Out-of-town customers

THE BREAKDOWN

1. Find the total sales for every customer in each store.
2. Attach the city and state for the store in which each purchase was made.
3. Join the city and state information from every customer's address and use this to find out-of-town customers.
4. Roll up to:
    a. Find and attach the total sales for and total number of out-of-town customers.
    b. Find the total sales for and total number of customers in each store.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W